Solucionario de la Olimpíada Informática Argentina 2019

Autores

Brian Bokser

Sebastián Cherny

Agustín Santiago Gutiérrez

Facundo Gutiérrez

Lautaro Lasorsa

Carlos Miguel Soto

Ariel Zylber

Índice general

1 Introducción **1**

2 Selectivo para la IOI **3**

2.1. Día 1 . . . 3

2.1.1. Problema 1: Buscando parejas [bailando] . . . 3

2.1.2. Problema 2: Organizando el depósito [deposito] . . . 7

2.1.3. Problema 3: Comprando adaptadores [multicontactos] . . . 15

2.2. Día 2 . . . 18

2.2.1. Problema 1: Cultivando bacterias [bacterias] . . . 18

2.2.2. Problema 2: ¡Jugando rápido! [speedrun] . . . 24

2.2.3. Problema 3: Recuperando distancias [distancias] . . . 28

3 Certamen Jurisdiccional **37**

3.1. Nivel 1 . . . 37

3.1.1. Problema 1: Programando calculadoras [calculadora] . . . 37

3.1.2. Problema 2: Organizando el Librero [librero1] . . . 38

3.1.3. Problema 3: Preparando la Receta [receta] . . . 39

3.1.4. Problema 4: Enfrentando Monstruos [batamon] . . . 41

3.2. Nivel 2 . . . 49

3.2.1. Problema 1: Tetris de Plástico [plastetris] . . . 49

3.2.2. Problema 2: Preparando Recetas [recetas] . . . 53

3.2.3. Problema 3: Organizando el Librero [librero2] . . . 54

3.2.4. Problema 4: Aprovechando Comodines [comodines] . . . 54

3.3. Nivel 3 . . . 57

3.3.1. Problema 1: Jugando Generala [generala] . . . 57

3.3.2. Problema 2: Organizando el Librero [librero3] . . . 60

3.3.3. Problema 3: Ubicando fichas en la hilera [hilera] . . . 63

3.3.4. Problema 4: Ubicando la oficina de correo [correocentral] . . . 70

4 Certamen Nacional **77**

4.1. Nivel 1 . . . 77

4.1.1. Problema 1: Aprendiendo operaciones [aprendiendo] 77
4.1.2. Problema 2: Computando patentes [patentes] 80
4.1.3. Problema 3: Caminando al Colectivo [colectivo] 89
4.2. Nivel 2 92
4.2.1. Problema 1: Escalera al cielo [escalera] 92
4.2.2. Problema 2: Aplicando operaciones [aplicando] 95
4.2.3. Problema 3: Truco de magia [mago] 99
4.3. Nivel 3 101
4.3.1. Problema 1: Dominando operaciones [dominando] 101
4.3.2. Problema 2: Laboratorio lleno de perros [laboratorio] 102
4.3.3. Problema 3: Ayudando al Electricista [electricista] 105

Capítulo 1

Introducción

Todos los enunciados de los problemas se encuentran disponibles en:
http://www.oia.unsam.edu.ar/problemas-categoria-programacion

Además, se pueden realizar envíos de soluciones para los problemas en el juez online de la olimpíada http://juez.oia.unsam.edu.ar. Se puede entrar directamente la página de un problema particular accediendo a http://juez.oia.unsam.edu.ar/#/task/PROBLEMA/statement, reemplazando el texto `PROBLEMA` por el "código de problema" correspondiente (el nombre corto: `bailando`, `bacterias`, `speedrun`, etc).

Capítulo 2

Selectivo para la IOI

2.1. Día 1

2.1.1. Problema 1: Buscando parejas [bailando]

http://juez.oia.unsam.edu.ar/#/task/bailando/statement

Una vez modelado, este problema se corresponde en realidad con la subtarea en una dimensión del problema “Respondiendo pedidos de Radiotaxi” (radiotaxi) del Certamen Nacional 2018. Se puede buscar dicha explicación en el Solucionario 2018, para tener un punto de vista alternativo para las mismas ideas, que complemente lo aquí explicado.

2.1.1.1. Solución por búsqueda exhaustiva

Una forma válida - aunque ineficiente - para resolver el problema es utilizar un método de búsqueda exhaustiva. Por ejemplo, podemos utilizar el método de fuerza bruta: enumerar absolutamente todas las posibilidades, evaluar el puntaje de cada una, y devolver el mínimo de todos los puntajes obtenidos.

En este caso, una “solución” consiste en una asignación o correspondencia, que a cada uno de los F famosos le asigne uno de los bailarines, sin repetir. Pueden quedar bailarines sin asignar, pero por cada famoso hay que asignar un bailarín.

Si numeramos por ejemplo los bailarines de 1 a B, una forma posible muy sencilla de enumerar todas las posibilidades es generar todas las permutaciones posibles de los números entre 1 y B. Cada una de ellas nos da una asignación válida: asignamos al primer famoso con el primer bailarín de la permutación, al segundo famoso con el segundo bailarín, y así siguiendo hasta el famoso número F.

Notar que la misma asignación aparece en forma de muchas permutaciones

diferentes, pues no importa cómo ordenemos los sobrantes $B - F$ bailarines. No es necesario optimizar esto para resolver la subtarea más pequeña: Como la cantidad de permutaciones exploradas es $1 \cdot 2 \cdot 3 \cdots (B-1) \cdot B = B!$, y 10! es aproximadamente 3,6 millones, este método alcanza.

Explorar las permutaciones tiene la ventaja de que puede escribirse cómodamente en forma recursiva, o aún más práctico y eficiente, aprovechando la ya existente función `next_permutation` de C++ (http://wiki.oia.unsam.edu.ar/cpp-avanzado/algorithm/next-permutation).

Esta solución obtendría 8 puntos.

2.1.1.2. Solución con programación dinámica

Una observación clave que podemos realizar es que nunca conviene armar pares de parejas "cruzadas". Más precisamente: Consideramos una pareja entre un famoso de altura f_1 y un bailarín de altura b_1, y otra segunda pareja de alturas f_2 y b_2. Si tenemos $f_1 \leq f_2$, entonces queremos que sea $b_1 \leq b_2$. Sino, si fuera $b_1 > b_2$, podemos intercambiar bailarines y armar las parejas (f_1, b_2) y (f_2, b_1). Se puede comprobar en este caso que la diferencia máxima no aumenta al reordenar las parejas (en el problema radiotaxi, esto se corresponde en forma exacta al argumento por el cual nunca conviene que dos taxis se crucen).

Como consecuencia de lo analizado, se deduce que, si tan solo tuviéramos ya elegidos a los F bailarines que participarán del concurso, la forma óptima de asignarlos a los F famosos es simplemente en orden: el menor con el menor, el segundo menor con el segundo menor, y así siguiendo.

Teniendo esto en cuenta, podemos ordenar ambos arreglos - el de bailarines, y el de famosos - y a partir de ahora asumir que el bailarín i, de altura b_i, es el i-ésimo por altura, y lo mismo los famosos de alturas f_i.

Para resolver la parte de elegir cuáles de todos los bailarines queremos, podemos utilizar programación dinámica: podríamos imaginar un proceso donde vamos "viajando" por ambas listas a la vez, manteniendo índices i a los famosos y j a los bailarines, y en cada paso tenemos dos opciones: o formar la pareja entre el famoso i y el bailarín j, o saltear al bailarín j. Esto no se pierde soluciones porque justamente, los bailarines que se correspondan con los famosos los iremos encontrando en orden, por la observación anterior, así que si un bailarín no lo vamos a asignar inmediatamente con el siguiente famoso, podemos descartarlo.

Esto da lugar a una recursión: si llamamos $h(i, j)$ a la mínima diferencia posible de alturas entre bailarines y famosos si asignamos en forma perfecta a los famosos

de i hasta F con los bailarines de j hasta B, la respuesta al problema será $h(1,1)$ y podemos calcular los valores de h con la siguiente recursión:

$$h(i,j) = \begin{cases} 0 & \text{para } i = F+1 \\ -\infty & \text{para } i \leq F, j = B+1 \\ min(h(i,j+1), max(h(i+1,j+1), |f_i - b_j|)) & \text{sino} \end{cases}$$

La complejidad de esta solución es $O(FB)$. Esta solución implementada eficientemente obtendría 53 puntos.

2.1.1.3. Solución con búsqueda binaria + goloso

La solución esperada para este problema utiliza búsqueda binaria en la respuesta. Se puede leer sobre búsqueda binaria en http://wiki.oia.unsam.edu.ar/algoritmos-oia/busqueda-binaria.

Pasamos a analizar entonces cómo decidir, dado un valor X (que sería el valor que estamos probando en un cierto paso de la búsqueda binaria), si es posible emparejar a los famosos con bailarines de manera tal que ninguna pareja tenga una diferencia de alturas superior a X.

Lo que podemos pensar es que entonces un famoso de altura h_f puede emparejarse con cualquier bailarín cuya altura h_b satisfaga $h_f - X \leq h_b \leq h_f + X$. Dicho de otra manera equivalente, debe ser $h_b \in [h_f - X, h_f + X]$. Es decir, por cada famoso existe un intervalo de longitud $2X$, cuyos extremos están en $h_f - X$ y $h_f + X$. Si pensamos que cada bailarín es un punto en la recta numérica dado por su altura (pueden existir varios bailarines en un mismo punto, pues pueden tener la misma altura), la pregunta es si es posible a cada intervalo (famoso) asignarle algún punto (bailarín) contenido en él, y con la restricción de que un mismo punto (a menos que sean "múltiples copias" de esa altura correspondientes a diferentes bailarines) no puede utilizarse para cubrir diferentes intervalos.

Este tipo de problema donde hay intervalos y puntos que se tocan o cubren entre sí es un problema clásico. Como ocurre en muchos problemas de intervalos, es posible dar un algoritmo mucho más eficiente ordenando y utilizando una técnica de barrido.

La observación clave es que en este caso, siempre conviene al intervalo de más a la izquierda (En este problema particular todos tienen la misma longitud, pero en general, sería el que **termina primero**, es decir, el que tiene el extremo derecho

más a la izquierda que todos los demás) asignarle el punto de más a la izquierda posible, es decir, el punto de más a la izquierda entre todos los contenidos en ese intervalo.

Si realizamos este proceso repetidamente, iremos recorriendo los intervalos en orden, y en cada paso "consumimos" el punto de más a la izquierda dentro del intervalo, con lo cual **los puntos también los iremos recorriendo en orden de izquierda a derecha**. Esto permite mantener una variable que indique el índice del siguiente punto a considerar: cada vez que debemos buscar el punto de más a la izquierda contenido en el intervalo actual, simplemente descartamos puntos mientras queden más a la izquierda y afuera del intervalo. A continuación, si el siguiente punto existe (no se han agotado todos) y queda dentro del intervalo actual, ese es el que debemos utilizar para este intervalo. Si en cambio ese punto queda fuera del intervalo (pero a la derecha), llegamos a una situación en la cual no existe ningún punto contenido dentro del intervalo actual, y por lo tanto ya es imposible realizar la asignación, dadas las asignaciones de puntos que ya realizamos.

Este goloso no puede tener nunca falsos positivos: si dice que para un cierto X es posible asignar, debe ser posible porque el algoritmo constructivamente realiza una asignación válida para ese X. Para demostrar que este goloso es correcto, lo que hay que demostrar es que nunca puede haber un falso negativo: es decir, que si este método no consigue una asignación válida con diferencia máxima X, es necesariamente porque ninguna existía.

Para demostrar eso, aplicamos una técnica extremadamente común y útil para demostrar que un algoritmo goloso es correcto: probaremos que **en cada paso, la decisión del algoritmo es segura**. Esto quiere decir que, **suponiendo que existe una solución antes de un cierto paso**, demostraremos que **sigue existiendo después de realizar un paso**. Con probar esto alcanza porque entonces si existe solución al comenzar, como el algoritmo solo hace pasos seguros, nunca puede fallar y perderse de encontrar alguna solución.

Supongamos entonces que un cierto paso no fue seguro. Esto quiere decir que el algoritmo eligió para el intervalo de más a la izquierda, el menor punto posible, pero al hacerlo se perdió la solución. Es decir, existía una solución, pero no existe ninguna donde se asigne al primer intervalo a ese primer punto p_1. Consideramos entonces el punto p_2 que esa solución correcta que sí existe le asigna el primer intervalo. Debe ser $p_1 \leq p_2$ necesariamente porque p_1 es el menor posible de todos los puntos asignables al intervalo. Si en esta solución hipotética que existe, el punto p_1 quedó libre y no se usó con ningún intervalo, entonces sigue siendo solución si dejamos libre p_2 y usamos en su lugar p_1 para cubrir al primer intervalo, así que el paso en realidad hubiera

sido seguro.

El único caso que resta por analizar es aquel en que en la solución de referencia considerada, p_2 fue asignado al primer intervalo, pero p_1 en cambio fue asignado a algún otro intervalo i_2. La clave es que i_2 necesariamente termina más a la derecha (o en la misma posición) que el primer intervalo. Esto significa que i_2 abarca necesariamente todos los puntos que hay entre p_1 y el fin del primer intervalo. Pero p_2 pertenece al primer intervalo, así que está antes que el fin del primer intervalo, y al ser $p_1 \leq p_2$, tenemos que por lo tanto i_2 contiene a p_2. En este caso tenemos entonces que ambos intervalos contienen a ambos puntos, y por lo tanto podríamos modificar nuestra solución sin problemas para que sea p_1 quien cubre al primer intervalo, y p_2 quien cubre a i_2. De esta manera concluimos que en este caso también existe una solución en la cual se usa p_1 para cubrir el primer intervalo, completando la demostración de que el primer paso es seguro. Este mismo razonamiento aplica a todos los pasos del algoritmo.

La complejidad de cada paso resulta $O((F + B) \lg(F + B))$, simplemente por ordenar el arreglo. Hay $O(\lg(h_{max} - h_{min} + 1))$ pasos gracias al método de búsqueda binaria. El algoritmo propuesto para el problema de asignar intervalos a puntos funciona correctamente incluso si los intervalos tienen diferentes longitudes (pues nunca utilizamos tal hecho en la demostración), pero algo que podemos aprovechar para este problema es que, al tener todos exactamente la misma longitud $2X$, ordenarlos por extremo derecho es lo mismo que ordenarlos por su centro, y su centro está justamente en las alturas de los famosos, que es algo que no cambia en ningún paso de la búsqueda binaria (que solamente mueve el valor X). Por lo tanto, en lugar de ordenar en cada paso, es posible ordenar una única vez al principio del algoritmo, y luego cada paso de la búsqueda binaria tendrá un costo $O(F + B)$ por recorrer los arreglos ya ordenados.

Esta solución obtendría 100 puntos.

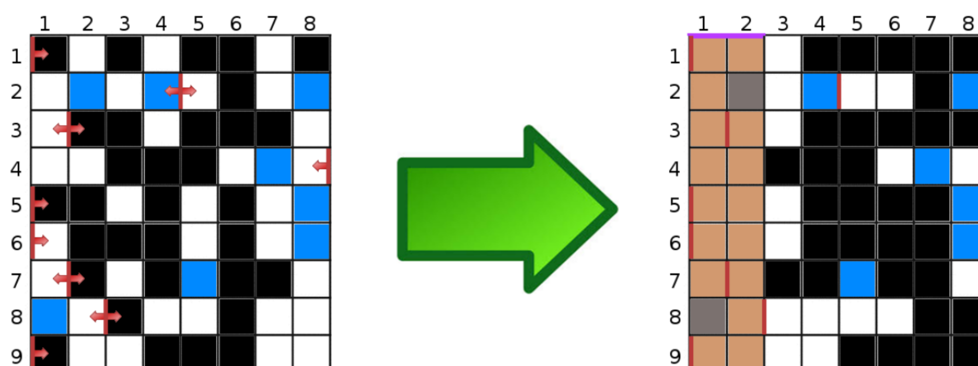
2.1.2. Problema 2: Organizando el depósito [deposito]

http://juez.oia.unsam.edu.ar/#/task/deposito/statement

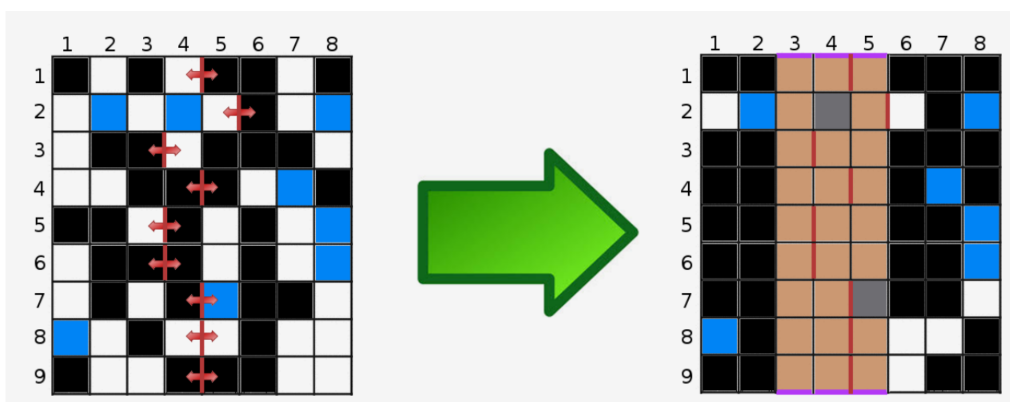
Descripción del problema

En el problema se nos describe un cierto depósito como una grilla de N filas y M columnas. Cada casilla puede tener una *baratija* (hay T de ellas), una *gotera* (hay G de ellas) o estar *vacía*. En cada fila se puede ubicar una *pinza* antes o después de cada casilla. Las pinzas actúan hacia cada lado moviéndose cada una hacia un

extremo distinto de la fila y barriendo todas las baratijas a su paso como se ve en la figura.



Ninguna baratija, ni ninguna pinza puede tocar una gotera a su paso, por eso la pinza se detiene instantes antes de que esto ocurra. El objetivo del problema es ubicar las pinzas en el diseño del depósito de forma de generar un rectángulo de $N \times L$ de casillas sin baratijas con el mayor L posible. En la figura anterior vimos cómo generar una solución con $L = 2$ para el diseño de la figura. A continuación se muestra una posible forma de ubicar las pinzas, con la cual se obtiene una solución con $L = 3$ (es decir, un rectángulo libre de baratijas de $N \times 3$).



Subtarea 1 ($N = 1, M \leq 10^6, G \leq 10^5, T \leq 10^5$)

En esta subtarea el diseño del depósito tiene **una sola fila**, y hay a lo sumo 10^6 columnas. Entre cada par de goteras, la cantidad de baratijas y de casillas vacías está fija, por lo tanto, en cualquier lugar que ubiquemos las pinzas, entre cada par de goteras se generará la misma cantidad de espacios libres al ubicar una pinza en el intervalo.

Con este razonamiento, si uno analiza solo localmente, parece no importar dónde ubicamos la pinza en el intervalo, **pero globalmente sí importa**, pues

dependerá de los espacios que quedan libres a cada lado de una gotera (recordar que podemos ubicar el barco sobre casillas con goteras). Veamos la siguiente situación.

1	2	3	4	5	6	7	8

Ubicando una pinza en las primeras 4 posiciones posibles, obtenemos las siguientes disposiciones además de la original (descartando casos donde quedan disposiciones repetidas).

1	2	3	4	5	6	7	8

1	2	3	4	5	6	7	8

¿En todas se obtiene la misma respuesta? **No**, en el primer y segundo caso se obtiene $L = 2$, sin embargo en el último caso se obtiene $L = 4$, pues las casillas $3, 4, 5$ y 6 quedan sin baratijas y se puede ubicar el bote allí. ¿Qué ocurrió? Nuestro análisis local fue correcto, entre las casillas $1, 2, 3$ y 4 siempre quedaron exactamente 2 casillas sin baratijas, pero en el último caso, al dejar las casillas libres aledañas a la gotera, se puede aprovechar el espacio sin baratijas que está del otro lado de la gotera.

De este análisis se desprende que *los únicos lugares candidatos a ubicar una pinza es a cada lado de cada gotera*. Pues ubicando una pinza en cualquier otro lugar del intervalo, obtendríamos una solución que no empeora al valor que se obtiene ubicando una pinza en cualquiera de los lugares aledaños a las goteras más cercanas (o extremo del tablero).

Entonces, ¿qué debemos calcular para obtener la respuesta? Para cada gotera, podemos calcular 2 valores a cada lado (si no hay goteras, la respuesta es simplemente la cantidad de casillas sin baratijas).

- Cantidad de casillas hasta la primera baratija a *izquierda*.
- Cantidad de casillas hasta la primera baratija a *derecha*.
- Cantidad de casillas sin baratijas hasta que las baratijas empujadas por una pinza se topen con una gotera a *izquierda*.

- Cantidad de casillas sin baratijas hasta que las baratijas empujadas por una pinza se topen con una gotera a *derecha*.

Las primeras dos cantidades indican la cantidad de casillas que tenemos a un lado hasta toparnos con la primera baratija (puede haber goteras en el medio), que es lo que aportará ese lado a la respuesta si no ubicamos una pinza a ese lado de la gotera. Las últimas dos cantidades indican la cantidad de casillas libres que tenemos a un lado luego de ubicar una pinza en ese lado. Debemos tener cuidado en que una pinza que pasó por una gotera no empujará baratijas, en cuyo caso ambas cantidades para esa dirección coincidirán.

Una vez calculados estos números, que llamaremos `sin_pinza_izq`, `sin_pinza_der`, `con_pinza_izq`, `con_pinza_der` respectivamente, simplemente debemos calcular para cada gotera:

$$\max\{\texttt{sin_pinza_izq} + \texttt{con_pinza_der}, \texttt{con_pinza_izq} + \texttt{sin_pinza_der}\}$$

Finalmente, debemos devolver el mayor valor obtenido entre todas las goteras. Para calcular ambas cantidades puede ser conveniente hacer dos iteraciones, una de izquierda a derecha y otra de derecha a izquierda, cosa de resolver el problema en tiempo lineal en la cantidad de columnas, es decir, $\mathcal{O}(M)$.

Subtarea 2 ($N \leq 10^3, M \leq 10^6, G = 0, T \leq 10^5$)

En esta subtarea **no hay goteras**. Si en la fila i sabemos que hay c_i casillas libres, entonces la solución necesariamente será menor o igual a c_i, pues si el bote midiese $L > c_i$ no entraría en la fila i.

Por lo tanto, sabemos que en este caso la solución al problema necesariamente cumple que $L \leq \min\limits_{1 \leq i \leq N} c_i$. ¿Podemos lograr esta cota? Sí, podemos. Por ejemplo, una forma de lograr una solución con $L = \min\limits_{1 \leq i \leq N} c_i$ sería ubicar todas las pinzas en la primera ubicación disponible, empujando en cada fila a todas las baratijas hasta el final de dicha fila.

Para calcular eficientemente la solución, podemos tener un contador de casillas libres en cada fila que comienza en M para cada fila, y para cada baratija restar uno al contador en su fila correspondiente. De esta forma, la cantidad de operaciones realizada es $\mathcal{O}(N + T)$.

Subtarea 3 ($N \leq 10^3, M \leq 10^6, G \leq 10^5, T = 1$)

En esta subtarea **hay una sola baratija**, por lo tanto el depósito está prácticamente vacío. En las $N-1$ filas donde no hay baratijas claramente no importa dónde ubicamos la pinza, pues el bote puede ocupar cualquier posición (recordar que se puede ubicar por encima de goteras).

Por lo tanto, solamente debemos analizar qué ocurre en la fila donde se encuentra la baratija. Si ubicamos la pinza a la izquierda de la baratija, esta se moverá todo lo posible hacia la derecha hasta que la baratija se tope con una gotera o el borde del depósito (siempre y cuando no haya una gotera en el recorrido de la pinza hasta la baratija). De la misma forma, ubicando una pinza a la derecha de la baratija, esta se moverá todo lo posible hasta la izquierda hasta toparse con una gotera o el fin del depósito.

Utilizando el análisis anterior, concluimos que solamente hay dos casos a estudiar, según de qué lado de la baratija ubicamos la pinza. Ambos casos podemos simularlos de manera similar a la primera subtarea.

Subtarea 4 ($N \leq 100, M \leq 100, G \leq 10^5, T \leq 10^5$)

En esta subtarea el depósito es pequeño. Veamos cómo podemos aprovecharlo. Supongamos que fijamos la posición final del bote y nos preguntamos si efectivamente podemos ubicarlo allí. Es decir, para un par (i, j) con $1 \leq i \leq j \leq M$, veamos si podemos ubicar el bote en las columnas $i, \ldots, j$.

Una vez que tenemos fijado el intervalo, ¿cuándo vamos a poder? Necesitamos en **todas las filas** poder dejar las columnas $i, \ldots, j$ libres de baratijas. Por lo tanto, nos queda resolver el problema en una fila de ver si podemos dejar libres las columnas $i, \ldots, j$. Esto último podemos hacerlo de manera similar a la subtarea 1 en $\mathcal{O}(M)$.

Por lo tanto, para saber si podemos ubicar el bote en (i, j), iteraremos sobre todas las filas y diremos que es posible ubicar el bote allí si y solo si podemos efectivamente liberar de baratijas las columnas $i, \ldots, j$ en cada fila. Por lo tanto, la verificación de si podemos ubicar el bote en (i, j) nos lleva $\mathcal{O}(N \cdot M)$. En total hay $\mathcal{O}(M^2)$ pares (i, j) posibles, los cual nos daría una complejidad total de $\mathcal{O}(N \cdot M^3)$, que no es suficiente para completar esta subtarea.

Con esta misma idea, ¿cómo podemos hacerlo más eficiente? ¿Hace falta realmente mirar todos los pares (i, j)?. Aquí hay dos ideas muy relacionadas que nos pueden ayudar.

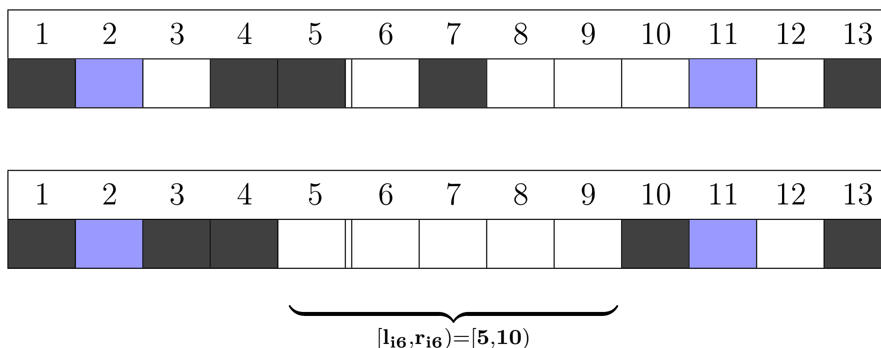
La primera es notar que si podemos ubicar un bote en el depósito de tamaño

$N \times L$, entonces podemos ubicar un bote de tamaño $N \times K$ con $K \leq L$. Por lo tanto, podemos hacer una *búsqueda binaria*[1] en la respuesta. Fijado un valor de L, podemos probar solamente los pares (i, j) de tamaño L, es decir los (i, j) tales que $j - i + 1 = L$. La clave está en que solo hay $\mathcal{O}(M)$ pares de tamaño L y solo estaremos probando $\mathcal{O}(\lg M)$ valores distintos de L. Por lo tanto, juntando estas dos ideas, podemos obtener una solución que solo realiza $\mathcal{O}(N \cdot M^2 \lg M)$ operaciones.

Realizando un análisis similar, podemos mejorar aún más esta idea utilizando la técnica de *two pointers*. Para cada posible comienzo i, llamemos j_i al mayor valor de j para el cual es posible ubicar un bote en las columnas $i, \ldots, j$. ¿Qué ocurre con j_{i+1}? Si podemos ubicar un bote en las columnas $i, \ldots, j_i$, entonces más aún podremos ubicarlo en $i + 1, \ldots, j_i$. Por lo tanto el valor de j_{i+1} (que si recordamos es el mayor j para el cual podemos ubicar un bote en $i, \ldots, j$) necesariamente será mayor o igual al valor de j_i. Es decir $j_{i+1} \geq j_i$, o equivalentemente, $\{j_i\}_{i=1}^{n}$ es una sucesión creciente. Por lo tanto, solo nos hace falta hacer la verificación para $\mathcal{O}(M)$ pares de columnas (i, j), realizando de esa forma $\mathcal{O}(N \cdot M^2)$ operaciones.

Subtarea 5 ($N \leq 10^3, M \leq 10^3, G \leq 10^5, T \leq 10^5$)

En cada fila, las $\mathcal{O}(M)$ posibles ubicaciones de las pinzas nos dan un intervalo que podemos obtener libre de baratijas. En una fila podemos obtener todos estos intervalos en $\mathcal{O}(M)$ haciendo dos pasadas (una de izquierda a derecha y otra de derecha a izquierda) de forma similar a la subtarea 1. Para la $j-$ésima posible ubicación de la pinza en la fila i, vamos a llamar $[l_{ij}, r_{ij})$ al intervalo que podemos dejar libre de baratijas en la j-ésima ubicación posible de una pinza en la fila i, como vemos en la siguiente figura.



La idea será ir recorriendo en orden estos posibles intervalos candidatos. Al comenzar, en cada fila tenemos el intervalo de posiciones libres de baratijas generado por ubicar en la primera posición posible de cada fila a una pinza, es decir, los intervalos de la forma $[l_{i0}, r_{i0})$. Podemos calcular la intersección de todos, dependerá

[1] http://wiki.oia.unsam.edu.ar/algoritmos-oia/busqueda-binaria

de $\max_{1 \leq i \leq N} l_{i0}$ y de $\min_{1 \leq i \leq N} r_{i0}$, y representa la mayor cantidad de posiciones libres que podemos tener en todas las filas para esta ubicación de las pinzas.

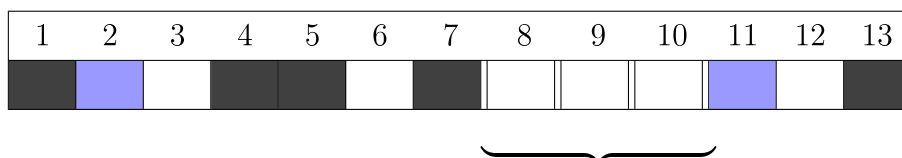
La idea será recorrer todos estos intervalos candidatos en orden, de forma de obtener en todo momento el tamaño de la intersección de estos intervalos. Pero la pregunta es ¿en qué orden?. Para ello debemos pensar ¿cuál nos está restringiendo el tamaño del bote?, **aquel con menor extremo derecho**, por lo tanto, en la fila donde se realice el menor extremo derecho, debemos pasar al siguiente intervalo candidato.

Así, podemos realizar una *técnica de barrido*, muchas veces llamada *sweep line*, e ir recorriendo los $\mathcal{O}(N \cdot M)$ intervalos candidatos. Es conveniente utilizar alguna estructura de datos que nos permita rápidamente obtener el mínimo de un conjunto, en los que guardaremos los extremos derechos (para saber el mínimo, que nos indica el próximo intervalo a salir) y en otro los extremos izquierdos (para saber el máximo, y así calcular en todo momento la longitud de la intersección de los intervalos). Para obtener la respuesta al problema debemos quedarnos con la máxima intersección encontrada a lo largo del barrido. En total, realizamos $\mathcal{O}(N \cdot M \lg(N))$ operaciones.

Subtarea 6 ($N \leq 10^6, M \leq 10^9, G \leq 10^5, T \leq 10^5$)

Solamente viendo las dimensiones del depósito podemos ver que no podremos representar a todas sus casillas, pues hay $N \cdot M \leq 10^{15}$ casillas, entonces aún guardando solo un bit en cada casilla estaríamos utilizando más de ¡100 tera bytes de memoria!.

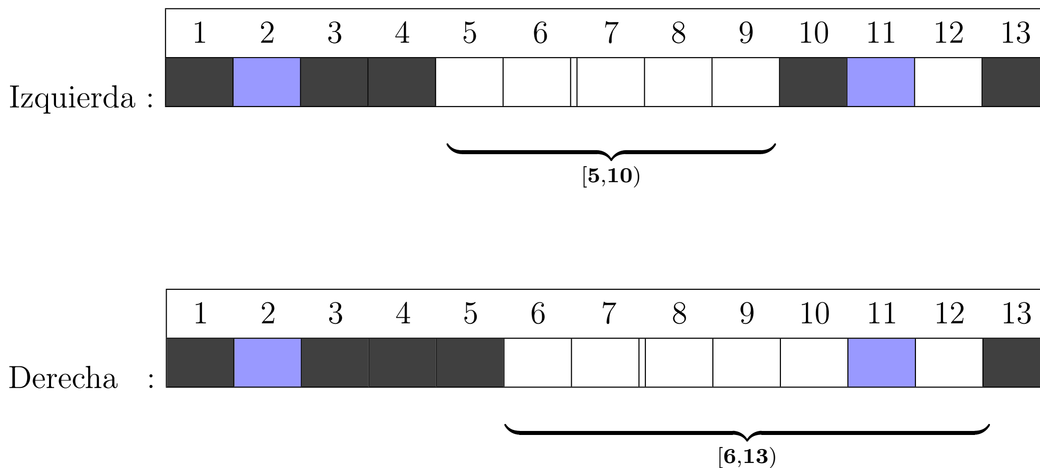
La observación clave para resolver esta subtarea es notar que de todas las posiciones candidatas a ubicar una pinza, hay muchas que arrojan la misma respuesta. Entonces, veamos cómo obtener una *discretización más fina en los candidatos.*



Observemos en el ejemplo anterior que ubicar una pinza entre las columnas $7-8$, entre las columnas $8-9$, entre las columnas $9-10$, o entre las columnas $10-11$ es completamente equivalente. Esto se debe a que no importan las casillas libres que hay en el medio entre baratijas y goteras. ¿Ahora hace falta que separemos a parte entre gotera y baratija? No, pues si ubicamos una pinza justo al costado de una gotera (en el ejemplo sería $10-11$), se obtiene el mismo resultado que ubicando

una pinza justo delante de la primera baratija en esa dirección (en el ejemplo sería $7-8$).

Por lo tanto, **los únicos lugares candidatos a ubicar pinzas es a cada lado de una baratija**. Notemos que si bien el depósito es grande, en esta subtarea la cantidad de baratijas T está acotada por 10^5. Para la i-ésima baratija, llamemos $[l_i^{izq}, r_i^{izq})$ al intervalo que queda libre de baratijas al ubicar una pinza a la izquierda de una baratija. Por ejemplo, si lo hacemos a la izquierda de la baratija en la séptima posición en el ejemplo, obtenemos el intervalo $[l_i^{izq}, r_i^{izq}) = [5, 10)$. Análogamente, ubicando a la derecha de la correspondiente baratija, obtenemos el intervalo $[l_i^{der}, r_i^{der} = [6, 13)$. Notar la notación *inclusive-exclusive* en los extremos.



Por lo tanto, en total, tendremos $\mathcal{O}(T)$ intervalos candidatos entre todas las filas. En cada fila con baratijas tenemos una lista de intervalos candidatos. Cada una de ellas podemos ordenarla por sus extremos derechos y calcular la intersección entre todos los intervalos (es decir, el tamaño máximo de bote que podemos ubicar con esta primera elección de pinzas). De todos estos intervalos, ¿cuál nos está restringiendo el tamaño del bote?, nuevamente **aquel con menor extremo derecho**, por lo tanto, en la fila donde se realice el menor extremo derecho, debemos pasar al siguiente intervalo candidato.

De esta forma, realizando con cuidado esta *técnica de barrido*, podemos ir manteniendo en todo momento la intersección de todos los intervalos candidatos. Simplemente debemos quedarnos con el tamaño del máximo intervalo durante el barrido. Calcular estos intervalos para cada baratija se puede hacer en $\mathcal{O}(N + T + G)$, y el barrido requiere un ordenamiento previo de los intervalos, lo cual lleva $\mathcal{O}\left((T+G) \cdot \lg(T+G)\right)$ operaciones y es suficiente para obtener un puntaje completo por el ejercicio.

2.1.3. Problema 3: Comprando adaptadores [multicontactos]

http://juez.oia.unsam.edu.ar/#/task/multicontactos/statement

Este problema es interesante pues aunque en un principio quizás no lo parezca, al modelarse adecuadamente resulta ser un problema de teoría de grafos.

Consideremos un grafo con M nodos, uno por cada tipo de enchufe existente. En el problema nos piden realizar N “transformaciones” de un tipo de enchufe en otro: tenemos que ser capaces, mediante adaptadores, de transformar desde el tipo del primer enchufe de la zapatilla OIA, hasta el tipo primer enchufe de la zapatilla pedida por IOI. También desde el tipo del segundo de OIA hasta el tipo del segundo de IOI, y así siguiendo. Estos requisitos de transformación pueden resumirse colocando N aristas dirigidas en nuestro grafo de M nodos: Por cada posición i entre 1 y N, colocamos una arista desde el nodo correspondiente al tipo del i-ésimo enchufe del multicontacto que tiene OIA, hasta el nodo correspondiente al tipo del i-ésimo enchufe del multicontacto pedido por IOI. Notar que el grafo que nos queda puede tener aristas repetidas y autoejes, aunque si queremos podemos descartarlos pues estos requisitos repetidos no aportan nada.

Algo importante a notar en este problema es que como hay solamente $N < 1{,}000{,}000$ enchufes, nunca necesitaremos comprar más de una bolsa de un cierto tipo, pues nunca utilizaremos más de N adaptadores **de un mismo tipo**. Es decir, podemos dar por sentado que comprar una bolsa con un cierto tipo de adaptador nos habilita a utilizar ese adaptador tantas veces como queramos: la respuesta al problema no cambia en este caso si en lugar de un millón de adaptadores por bolsa, se tuvieran infinitos adaptadores por bolsa.

Notemos que de la misma forma que modelamos los requisitos como aristas dirigidas entre tipos de enchufes, es completamente natural modelar como aristas dirigidas a los tipos de adaptadores: cada tipo de adaptador es exactamente una arista dirigida desde el tipo de enchufe de entrada, hasta el tipo de enchufe de salida. La pregunta entonces es: dado un conjunto de aristas X cualquiera que se corresponda con adaptadores que hemos decidido comprar, ¿Cuándo podremos satisfacer el requisito correspondiente a una cierta arista $u \to v$?

La única forma de transformar un tipo de enchufe en otro es apilando adaptadores, y en este apilamiento, el tipo de salida de uno debe coincidir con el tipo de entrada del siguiente. Esto hace que justamente si miramos las aristas correspondientes a los adaptadores usados para cumplir el requisito de transformar un enchufe, estas formarán un camino desde u hasta v. En otras palabras: un conjunto X de tipos de adaptadores (aristas) sirve, si para todos los N requisitos

(u, v) existe un camino de u hasta v utilizando únicamente aristas de X.

El problema de grafos es entonces el siguiente: dado un grafo dirigido con M nodos (los tipos de enchufes) y N aristas (los requisitos de transformar tipos de enchufes), dar un conjunto X **cualquiera** de aristas (¡Pueden no estar en el grafo de entrada! Ya que podemos comprar cualquier bolsa que queramos.) tal que para cada una de las N aristas (u, v), exista un camino de u a v usando las aristas de X, y tal que X tenga la menor cantidad de aristas posible.

Otra forma decir lo mismo sería: dado un grafo dirigido, debemos dar otro con los mismos nodos (pero las aristas que queramos) de tal manera que siempre que existía camino de u a v en el original, exista camino de u a v en el nuevo, y de modo tal que el nuevo tenga la menor cantidad de aristas posibles.

Este modelado facilita razonar, sobre todo en las subtareas más difíciles, pero no es completamente necesario para resolver el problema en las subtareas más simples.

2.1.3.1. Subtarea 1

En este caso se nos garantiza que el multicontactos IOI tiene todos los enchufes de tipo 1. Esto significa que en el grafo, todas las aristas son de la forma $(u, 1)$, es decir que todas van al nodo 1.

En este caso, por cada nodo $u \neq 1$ que aparezca en una de estas aristas, será necesario poner alguna arista de la forma (u, x) en el grafo final (pues sino, no se podría salir nunca de u, y no se podría llegar a 1). Pero poniendo una arista $(u, 1)$ por cada uno, vemos que no perdemos ningún camino, así que eso basta para resolver. Lo único que hay que hacer en este caso es entonces eliminar aristas duplicadas, y los loops $1 \to 1$ que hubiera. En otras palabras, la cantidad óptima resulta ser la cantidad de números distintos mayores que 1 presentes en el arreglo `a`.

Esta solución alcanza para obtener 4 puntos.

2.1.3.2. Subtarea 2

Este caso es un poco más complicado, pero también admite razonablemente un análisis manual de la situación. En este caso, los nodos 1 y 2 son especiales ya que son los únicos a los cuales llegan aristas. Podemos distinguir 3 casos:

1. No hay ninguna arista entre 1 y 2: En este caso, vemos que el grafo consiste en dos subgrafos idénticos a los de la subtarea 1 y completamente independientes

entre sí. Podemos entonces resolver cada uno por separado de la misma manera que antes.

2. Todas las aristas entre 1 y 2 van en la misma dirección. En este caso, si eliminamos como antes todos los duplicados y loops del grafo de entrada, obtenemos una solución óptima. Esto es porque, si llamamos n a la cantidad de nodos que son tocados por alguna arista (es decir, todos los nodos no aislados), vemos que si ignoramos direcciones en los caminos, debemos obtener un grafo conexo de n nodos, y eso requiere un mínimo de $n-1$ aristas, que es lo que estamos obteniendo en este caso.

3. Existen las aristas $1 \to 2$ y la $2 \to 1$. En este caso, no es posible utilizar menos que n aristas: si lo hiciéramos con menos, como tiene que quedar conexo estaríamos usando $n-1$, pero entonces las aristas (ignorando la dirección) formarían un árbol y sería imposible tener un ciclo, en particular sería imposible viajar de 1 a 2 y también de 2 a 1. Esto prueba que en este caso sí o sí necesitamos n aristas por lo menos. Pero hay una manera muy simple de garantizar todas las conexiones usando n aristas: simplemente formamos un ciclo entre los n nodos, y eso garantiza que se puede viajar de cualquier nodo a cualquier otro.

Como solamente hay que recorrer las aristas contando su situación respecto a los nodos especiales 1 y 2, estos casos pueden verificarse en tiempo lineal sin problemas.

Esta solución resuelve las dos subtareas que vimos y obtiene 17 puntos.

2.1.3.3. Subtareas 3,4,5

Veamos ahora el caso general, donde no tenemos restricciones para el grafo más allá de su tamaño.

Podemos aplicar un razonamiento análogo al que hicimos para la subtarea 2, pero sin concentrarnos en los nodos especiales 1 y 2 sino haciéndolo en general: primero que nada, separamos el grafo en las componentes débilmente conexas (es decir, las componentes conexas si ignoramos completamente la dirección de las aristas). Estas son completamente independientes entre sí y las resolveremos por separado. Ahora bien, para resolver una componente débilmente conexa de n nodos, tendremos que poner mínimamente $n-1$ aristas en la respuesta. Aquí hay dos casos entonces:

1. Si la componente débilmente conexa no tiene ningún ciclo dirigido, entonces es un DAG y admite un ordenamiento topológico (http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos-dirigidos/toposort). Si simplemente ponemos los nodos en este orden, y usamos $n-1$ aristas para conectarlos en el orden del camino, con esas $n-1$ aristas habremos cubierto todos los caminos posibles que existían originalmente, pues justamente en un ordenamiento topológico todas las aristas van a nodos posteriores del ordenamiento. En este caso basta entonces con poner $n-1$ aristas, y usar algún algoritmo de toposort para obtenerlas.

2. Si la componente débilmente conexa tiene algún ciclo dirigido, entonces no se puede con $n-1$ aristas, por el mismo argumento dado en la subtarea 2: el grafo subyacente sería un árbol, y nos queda imposible tener un ciclo dirigido. Pero con n aristas siempre es posible conectar todos los nodos entre sí con un ciclo, garantizando que existen todos los caminos.

Analizando cada componente y detectando si tiene ciclo o no, entonces, podemos resolver el problema: para las que no tengan ciclos, computamos un ordenamiento topológico y usamos $n-1$ aristas formando un camino siguiendo el ordenamiento topológico. Y para las que tengan ciclo, simplemente unimos todos sus nodos formando un gran ciclo simple, en cualquier orden.

Aproximadamente:

- La subtarea 3 permite soluciones de complejidad cúbica. Por ejemplo, detectar ciclos utilizando el algoritmo de Floyd-Warshall.

- La subtarea 4 permite soluciones de complejidad cuadrática. Por ejemplo, utilizar un algoritmo de toposort iterativo en N pasos que explora los mismos nodos repetidas veces, o un BFS/DFS con matriz de adyacencia.

- La subtarea 5 requiere soluciones más eficientes, de complejidad lineal o cercana. Basta para esto utilizar alguno de los algoritmos eficientes de toposort, y representar al grafo con listas de adyacencia.

2.2. Día 2

2.2.1. Problema 1: Cultivando bacterias [bacterias]

http://juez.oia.unsam.edu.ar/#/task/bacterias/statement

Este es un problema interactivo, donde el foco no está puesto principalmente en la complejidad temporal del programa (tiempo que tarda en ejecutarse), sino en la complejidad en términos de la **cantidad de mediciones realizadas por el programa**. Mientras menos mediciones realice el programa para obtener el resultado, mayor cantidad de puntos.

2.2.1.1. Opción más básica

La primera observación clave es que si para una medición, ponemos una temperatura A en el tubo de cierta especie de bacteria, entonces como sabemos que la temperatura máxima soportada siempre es un entero entre A y B, las bacterias de ese tubo sobrevivirán. En otras palabras, poniendo una temperatura A en un tubo podemos efectivamente "ignorarlo", pues el resultado de la medición dependerá solamente de los otros tubos.

Haciendo esto, podemos resolver cada tubo por separado: si queremos concentrarnos en un solo tubo, basta poner temperatura A en todos los demás, y entonces la respuesta de la medición dependerá únicamente de si la temperatura elegida para el tubo seleccionado supera la máxima que soporta esa especie de bacterias.

Con esto en mente, podemos ir subiendo la temperatura del tubo gradualmente, midiendo primero con temperatura $A+1$, luego con $A+2$ y así siguiendo, hasta que para una cierta temperatura x el resultado sea que no hay reacción. En ese caso, sabremos que $x-1$ es la temperatura máxima soportada por esas bacterias. Y si llegamos hasta temperatura B y aún así sobreviven, la temperatura máxima es B para esas bacterias.

Esta opción realiza unas 500.000 preguntas en peor caso, y no es suficiente para obtener un puntaje positivo, pero estas ideas fundamentales nos servirán para todas las demás propuestas.

2.2.1.2. Búsqueda binaria por cada especie de bacteria

Con la misma idea anterior, podemos enfocarnos en un tubo por vez, pero en lugar de utilizar una búsqueda lineal como explicamos antes, podemos encontrar la temperatura máxima utilizando búsqueda binaria (`http://wiki.oia.unsam.edu.ar/algoritmos-oia/busqueda-binaria`).

Este método podemos aplicarlo porque al probar con una cierta temperatura x, sabemos que si hubo reacción, la máxima soportada es **al menos** x, y ya descartamos los valores menores que x. Y si no hubo reacción, la máxima es **a lo sumo** $x-1$, y

ya descartamos x y todos los mayores.

Como hay $B - A + 1 \leq 128 = 2^7$ valores posibles para la temperatura, bastan 7 preguntas para descubrir la temperatura, lo cual aplicado a los 4,000 tubos da en total 28,000 mediciones.

Esto es suficiente para obtener 20 puntos en el problema. Si por alguna razón se implementa la búsqueda binaria o un método similar más ineficientemente y se realizan hasta 10 preguntas por tubo en lugar de las 7 óptimas, se obtienen 10 puntos.

2.2.1.3. Encontrando las bacterias especiales

En las soluciones anteriores no utilizamos un dato fundamental del problema: existen a lo sumo $K \leq 250$ especies de bacterias especiales, que son las únicas que pueden tener temperatura máxima distinta de B. Es decir, como N puede ser hasta 4,000, en casos grandes más del 90 % de las bacterias tendrán temperatura máxima B.

Como explicamos antes, si al realizar una medición con valor B las bacterias sobreviven, ya podemos estar seguros de que esa especie de bacterias tiene temperatura máxima B, y no se necesitan más mediciones para esa especie. Si con temperatura B las bacterias no sobreviven, podemos entonces utilizar 7 mediciones para descubrir la temperatura exacta con búsqueda binaria.

De esta manera, tendremos a lo sumo $7K$ mediciones por las varias búsquedas binarias, es decir a lo sumo 1750 mediciones. Las restantes mediciones las utilizaremos ahora para enfocarnos en el problema de encontrar cuáles son las especies de bacterias cuya temperatura máxima no es B: de esta manera, solo haremos mediciones con temperaturas A o B, según queramos incluir una especie en el conjunto a verificar, o no.

Una estrategia muy simple para encontrarlas es testearlas una por una: por cada una de las N especies, verificamos si soporta temperatura B. Aquellas (a lo sumo K) que no lo hagan serán las únicas que causarán un proceso de búsqueda binaria.

La cantidad de total de mediciones resulta en este caso $4{,}000 + 1{,}750 = 5{,}750$, que es suficiente para obtener 30 puntos.

2.2.1.4. Encontrando las bacterias especiales más eficientemente

Anteriormente, al analizar la temperatura en un único tubo de ensayo, inicialmente propusimos una solución con búsqueda lineal, probando todos los valores de temperatura hasta ver cuál funcionaba. Y tuvimos una gran ganancia al cambiar a búsqueda binaria.

Podemos aprovechar la misma técnica también para encontrar las bacterias especiales: en lugar de probar tubo por tubo para ver si es especial o no, podemos enfocarnos en buscar con búsqueda binaria **el primer tubo con bacterias especiales**.

Imaginemos que realizamos una medición donde ponemos temperatura B en los primeros i tubos, y A en todos los demás. Si las bacterias sobreviven, significa que en todos los primeros i tubos las bacterias resisten temperatura B. En otras palabras, **no hay especies de bacterias especiales en los primeros i tubos**.

Usando mediciones de este tipo, podemos ir preguntando entonces si existe algún tubo con bacterias especiales entre los primeros i, para distintos valores de i. Esto permite con búsqueda binaria encontrar el primer i tal que sí existan, y eso significará que justamente en el tubo i es donde por primera vez aparece una especie de bacterias especiales.

Como $N \leq 4096 = 2^{12}$, bastan 12 preguntas para encontrar el primer tubo con bacterias especiales. Una vez que lo encontramos, podemos repetir el procedimiento con los tubos restantes, ignorando los primeros i (es decir, poniéndoles temperatura A en las mediciones subsiguientes).

De esta forma, en total usaremos $12K \leq 3{,}000$ mediciones para encontrar las bacterias especiales.

Esto da un total de $3{,}000 + 1{,}750 = 4{,}750$ mediciones, lo que permite obtener 70 puntos.

2.2.1.5. Solución de 100 puntos esperada

Una forma de alcanzar los 100 puntos es utilizar un enfoque recursivo para ir buscando dónde están las bacterias especiales.

Escribiremos una función recursiva $f(i, j)$, que encuentre correctamente **todos** los tubos con bacterias especiales del intervalo $[i, j)$. Inicialmente la llamaremos como $f(0, N)$.

Si $j - i = 1$, el intervalo contiene un único tubo, y entonces hacemos una única

medición para ver si es especial: este es el caso base.

Si no, igualmente realizaremos una medición en la que ponemos todos los tubos del intervalo con temperatura B (y los de fuera del intervalo, con temperatura A). Si las bacterias sobreviven, no hay ningún tubo especial y la función puede retornar.

En el caso restante, la clave es que podemos definir $c = \lfloor \frac{i+j}{2} \rfloor$ y considerar las dos partes $[i, c)$ y $[c, j)$ en que se divide el intervalo original. Recursivamente, llamamos a $f(i, c)$ y $f(c, j)$ para encontrar las bacterias especiales en esos intervalos.

Dibujando el árbol binario completo de posibles llamadas, se puede observar que los niveles que corresponden a intervalos de longitud 16 o más tienen 511 nodos en total, y de los restantes 4 niveles se recorren a lo sumo $2K$ nodos por nivel. De esta manera en total nunca se utilizan más de $511 + 8K \leq 2{,}511$ preguntas para obtener las bacterias especiales. Sumando las búsquedas binarias, esto da un total de $2{,}511 + 1{,}750 = 4{,}261$ en peor caso, que es suficiente para obtener 100 puntos.

El código completo de esta solución queda:

```
vector<int> tubos;

int binary(int A,int B,int i) {
    // Invariante: Con a sobrevive, con b muere
    int a = A, b = B;
    while (b-a > 1)
    {
        int c = (a+b)/2;
        tubos[pos] = c;
        if (medir(tubos))
            a = c;
        else
            b = c;
    }
    return a;
}

void f(int N, int A,int B, int i, int j) {
    for (int index=i;index < j; index++)
        tubos[index] = B;
    if (medir(tubos))
```

```
        return;
    if (j-i == 1) {
        // Bacteria especial!
        tubos[i] = binary(A,B,i);
    }
    else {
        for (int index=i;index < j; index++)
            tubos[index] = A;
        int c = (i+j)/2;
        f(N, A, B, i, c);
        f(N, A, B, c, j);
    }
}

vector<int> bacterias(int N, int A,int B, int K) {
    tubos = vector<int>(N,A);
    f(N,A,B,0,N);
    return tubos;
}
```

Aunque no es necesario para el problema, surge una simple optimización de notar que, en el análisis de la cantidad de mediciones que se realizan, contabilizamos que en un peor caso se recorren completamente muchos niveles de la recursión, concretamente hasta llegar a intervalos de tamaño 16. Podemos entonces directamente llamar a $f(0,16)$, $f(16,32)$ y así sucesivamente, ahorrando esas llamadas iniciales. De esta forma la solución siempre encontraría la respuesta con 4,000 mediciones o menos.

2.2.1.6. Solución de 100 puntos encontrada por varios participantes

Notablemente, varios participantes encontraron una solución muy simple y más efectiva que el enfoque recursivo anterior.

La idea consiste en separar los tubos de ensayo en bloques de longitud d: tendremos los tubos $[0,d)$, los $[d,2d)$, los $[2d,3d)$ y así siguiendo. En cada bloque, de la forma que ya explicamos para las soluciones anteriores, realizamos una medición para verificar si contiene algún tipo de bacteria especial.

Hecho esto, tendremos a lo sumo K bloques en los cuales hay bacterias especiales. Para cada uno de ellos, simplemente hacemos búsqueda lineal, preguntando por cada tubo individual para ver si es especial. El total de mediciones

así realizadas en el peor de los casos será $\left\lceil \frac{4000}{d} \right\rceil + 250d$. Esta cantidad depende del valor de d, así que podemos elegirlo para que la cantidad de preguntas sea lo menor posible. Experimentando valores, el óptimo es $d = 4$, lo que permite en a lo sumo 2,000 preguntas identificar todas las especies de bacterias especiales.

Sumando las búsquedas binarias, son en total $2{,}000+1{,}750 = 3{,}750$ mediciones, y se obtienen 100 puntos.

2.2.1.7. Solución aún mejor

Aunque el problema en la prueba no exigía seguir optimizando la cantidad de mediciones, combinando varias de las ideas anteriores, podemos obtener una solución aún mejor.

Concretamente, dividamos nuevamente en bloques pero con tamaño $d = 8$. Consultamos cada bloque, así que tendremos a lo sumo 250 bloques de tamaño 8 en los que encontrar sus tubos especiales.

La clave es que en estas condiciones, podemos encontrar cada uno de los (a lo sumo) 250 tubos especiales en sólo 4 preguntas: para eso utilizamos la técnica de búsqueda binaria explicada anteriormente para la solución de 70 puntos.

Si tenemos un cierto bloque en el cual sabemos que hay al menos un tubo especial, como los bloques miden a lo sumo 8, usando búsqueda binaria podemos en sólo 3 preguntas identificar el primero de esos tubos. Si resulta ser el i-ésimo del bloque, nos queda que en los tubos $j > i$ no sabemos si hay algún especial o no. Usando una pregunta más podemos saber si alguno de ellos es especial: si alguno lo es, esto ha creado un nuevo bloque donde sabemos que hay algún tubo especial. Y sino, hemos ya consumido por completo el bloque original.

En cualquier caso, la situación es que luego de 4 mediciones, hemos encontrado necesariamente un tubo con bacterias especiales, y en todos los bloques que nos quedan tenemos la seguridad de que existe algún tubo especial. Por lo tanto nos quedaremos sin bloques para procesar y habremos encontrado todas las especies de bacterias especiales luego de a lo sumo $4K \leq 1{,}000$ mediciones.

Como tenemos 500 bloques de 8, inicialmente hicimos 500 mediciones. Sumando todo esto y las búsquedas binarias finales para saber las temperaturas especiales, la cantidad total queda $500 + 1{,}000 + 1{,}750 = 3{,}250$.

2.2.2. Problema 2: ¡Jugando rápido! [speedrun]

http://juez.oia.unsam.edu.ar/#/task/speedrun/statement

Este problema es una variación sobre el clásico problema del viajante de comercio (http://wiki.oia.unsam.edu.ar/algoritmos-oia/problemas-generales/travelling-salesman-problem), combinándolo además con el problema de buscar caminos mínimos en grafos (http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/floyd-warshall).

Este es un problema principalmente "técnico", donde para un participante que conozca bien las soluciones y algoritmos conocidos para el problema del viajante de comercio, la dificultad central será entender y modelar en el programa cuidadosamente los diferentes requisitos y particularidades que se describen en el enunciado.

Sin embargo, hay varias subtareas de menor dificultad, que explicaremos primero.

2.2.2.1. Subtarea 2

En esta subtarea, todas las tareas deben realizarse en el sitio 0, que es el sitio inicial donde comenzamos el juego. En este caso entonces, no tiene ninguna utilidad desplazarse a otros sitios utilizando los senderos, y lo único que hay que hacer es realizar tareas, en un orden válido, hasta realizar la tarea 0. Aquellas tareas que no sean necesarias para poder cumplir con la tarea 0, no las realizaremos.

Una observación clave es que, **al no tener que viajar**, el tiempo total depende únicamente del conjunto de tareas que realizamos, pero **no depende del orden** en que realicemos las tareas. Esta es la diferencia clave entre esta subtarea y las subtareas 4,5 y 6

Podemos imaginar un grafo de **dependencias entre tareas**: Cada vez que la tarea i es requisito para realizar la tarea j, podemos poner una arista dirigida desde la tarea i hasta la tarea j. De esta manera, solamente nos interesa realizar aquellas tareas u tales que exista un camino desde u hasta la tarea 0, pues son las que "bloquean" la realización de la tarea 0: las demás tareas podemos simplemente descartarlas.

Siempre que tenemos un grafo de dependencias, podemos obtener un orden válido para realizar las operaciones respetando las dependencias utilizando un algoritmo de ordenamiento topológico (http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos-dirigidos/toposort). De esta forma, la solución consiste en realizar las tareas necesarias en un orden topológico cualquiera del DAG de dependencias (habiendo descartado aquellas tareas que no conducen a la tarea 0: ¡Notar que el grafo de dependencias puede tener ciclos! Siempre y cuando esos ciclos

no conduzcan a la tarea 0).

2.2.2.2. Subtarea 3

En esta subtarea, la tarea 0 requiere solamente la 1, la tarea 1 requiere solamente la 2, y así hasta la tarea $T-1$ que no requiere a ninguna otra (pues si a su vez la $T-1$ requiriese alguna otra, sería imposible ganar el juego, pues no existe ninguna tarea inicial que se pudiera realizar).

Como el grafo de dependencias forma un camino simple, esto quiere decir que hay una única estrategia bien definida de tareas a realizar (En los términos de la subtarea 2, existe un único ordenamiento topológico): Primero hay que realizar la tarea $T-1$, luego la $T-2$, y así siguiendo hasta realizar finalmente la tarea 0 y ganar el juego.

Para implementar esto, en todo momento, guardamos en una variable el sitio actual en el que se encuentra el protagonista (inicialmente, el sitio 0). Y vamos iterando descendentemente las tareas desde $T-1$ hasta 0, y lo que hacemos es viajar desde el sitio actual hasta el sitio de esa tarea por un camino óptimo. Podemos computar estos caminos óptimos utilizando cualquiera de los algoritmos clásicos de camino mínimo: Floyd-Warshall, Bellman-Ford o Dijkstra.

Hay que tener el cuidado extra de no colocar en la respuesta final de acciones, la realización de la tarea 0. Lo más simple implementativamente es colocar ese 0 final igualmente, para no separar el caso en el código, pero removerlo justo antes de retornar la respuesta final.

2.2.2.3. Matriz de distancias

Para resolver todas las subtareas que siguen, será ampliamente conveniente tener ya computada la **matriz de distancias** del grafo de senderos. Esto podemos computarlo una única vez al comienzo del programa utilizando el algoritmo de Floyd-Warshall, y de esa manera ya sabremos para cualquier par de sitios i y j, el tiempo que insume viajar desde i hasta j en forma óptima (y podremos reconstruir un camino para hacerlo, cuando sea necesario).

2.2.2.4. Subtarea 4

En esta subtarea no hay ninguna restricción al grafo de dependencias, y cualquier tarea puede depender de cualquier otra. Notemos que si ya supiéramos de alguna manera en qué orden queremos llevar a cabo las T tareas, podemos aplicar el mismo método exacto de la subtarea 3 para realizarlas en ese orden, y obtendríamos el tiempo mínimo para realizar las tareas **en ese orden**.

Una solución para la subtarea 4 es probar los $T!$ posibles ordenamientos, y por cada uno calcular su costo como se hizo en la subtarea 3, tomando finalmente el mejor de todos. Además, puede verificarse durante la ejecución si un cierto ordenamiento no es válido, verificando que cuando se realiza una tarea ya estén realizadas todas las que requiere. Esto permite descartar los ordenamientos inválidos.

Esta solución de fuerza bruta implementada eficientemente es suficiente para resolver la subtarea.

2.2.2.5. Subtarea 1

En esta subtarea, las listas de requisitos son vacías, excepto para la tarea 0. Eso significa que desde el comienzo ya puede realizarse cualquier otra tarea, sin necesidad de haber realizado otras tareas antes, y la única restricción es estar presente en el sitio correspondiente a la tarea que se quiere realizar.

No conviene entonces en este caso realizar ninguna tarea que no sea la 0 y aquellas que necesita en forma directa, pues solo se sumaría tiempo adicional y no aportaría nada, al no permitir ninguna otra acción. La forma óptima de jugar en este caso es desplazarse para pasar por todos los requisitos de la tarea 0 lo más rápido posible, iniciando desde el sitio 0, y luego finalmente desplazándose hasta el sitio que corresponde a la tarea 0 para finalmente realizar la tarea 0.

Si reducimos el grafo de tal manera que solamente consideramos que existen los nodos correspondientes al sitio 0, el sitio de la tarea 0, y los sitios con las tareas necesarias, tomando como distancias entre ellos la distancia dada por la matriz de Floyd-Warshall, lo que tenemos será entonces un problema del viajante de comercio en forma exacta, iniciando en el sitio 0 y terminando en el sitio de la tarea 0. Podemos aplicar entonces la solución a este problema conocido, sin la complicación adicional de los requisitos y dependencias entre tareas.

2.2.2.6. Subtarea 5 y 6

La subtarea 5 admite buenos backtrackings que se ejecuten rápidamente, o implementaciones muy ineficientes de la idea de la solución esperada, al tener una cota intermedia entre las subtareas 4 y 6. Pasamos a describir la solución esperada para la subtarea 6, que corresponde al caso general y que por lo tanto resuelve también todos los anteriores.

La idea es adaptar levemente el algoritmo de programación dinámica usual para el problema del viajante de comercio. La solución al problema del viajante de comercio (TSP: Travelling Salesman Problem) se basa en plantear una recursión

$f(u, S)$ que indica el costo mínimo para recorrer todas las ciudades, si empezamos en la ciudad u y ya tenemos recorridas las ciudades del conjunto S. En nuestro caso plantearemos esto mismo, pero en lugar de ciudades, u será la tarea inicial y S serán las tareas ya realizadas.

En la recursión habitual para el problema del viajante de comercio, es posible considerar moverse desde u hasta cualquier otra ciudad v aún no recorrida, lo cual agregaría v al listado de ciudades ya recorridas [2]. En este caso podemos hacer lo mismo con las tareas, pero agregando un chequeo de modo que solamente esté permitido viajar a realizar una tarea v cuyos requisitos ya estén todos satisfechos en S. Para verificar eficientemente que todos los requisitos de v están en S, es conveniente tener representados tanto S como los conjuntos de requisitos mediante máscaras de bits, ya que entonces se hace con una simple operación `(requisitos[v] | S) == S` (http://wiki.oia.unsam.edu.ar/cpp-avanzado/operaciones-de-bits).

El detalle adicional es que cuando hablamos de movernos desde una tarea u a una tarea v, el costo será el tiempo que insume realizar la tarea u [3] más el costo de viajar desde el sitio de la tarea u hasta el sitio de la tarea v, que puede saberse en $O(1)$ al tener ya computada la matriz de distancias del grafo de sitios y senderos.

La complejidad final de calcular el tiempo mínimo resulta ser la suma de las complejidades usuales para Floyd-Warshall y para el problema de viajante de comercio, $O(N^3+T^2 2^T)$. Además, el camino óptimo puede reconstruirse combinando la reconstrucción de TSP con programación dinámica, con la reconstrucción de Floyd-Warshall, en un tiempo total adicional $O((T + N)T)$.

2.2.3. Problema 3: Recuperando distancias [distancias]

http://juez.oia.unsam.edu.ar/#/task/distancias/statement

Este es un problema output-only, donde conocemos todos los pares de distancias entre un cierto conjunto de puntos en la recta real, y nuestra misión es reconstruir todas las posibles soluciones existentes, es decir, todas las posibles configuraciones de N puntos en el plano que reproduzcan las distancias dadas (son $N + 1$ puntos incluyendo el punto 0, que se sabe que siempre está presente en el conjunto).

La particularidad de los problemas output-only es que se reciben todos los

[2] O bien agregaría u, dependiendo de si la convención usada es que S incluye a la ciudad inicial o no. Ambas pueden usarse indistintamente, mientras la implementación utilice una consistentemente todo el tiempo.

[3] O la v, dependiendo de la convención en uso.

archivos de entrada, y se debe enviar solamente las salidas, no necesariamente el código. Esto permite mirar los archivos de entrada manualmente, y utilizar cualquier herramienta, como por ejemplo programar en python, utilizar un programa de hoja de cálculo disponible en la computadora, utilizar programas de backtracking y dejarlos ejecutando durante 2 horas durante la prueba, o incluso resolver el caso en forma manual.

Si bien en rigor tenemos 10 subtareas exactamente, pues cada caso se evalúa por separado, se nos indican algunos grupos de casos que en este caso llamamos subtareas. Los analizamos a continuación:

2.2.3.1. Subtarea 1

En esta subtarea, que corresponde al primer archivo de entrada, se nos indica que $N = 5$, es decir que hay solamente 5 puntos desconocidos y este es el caso de entrada más pequeño de los 10. Si lo abrimos y vemos su contenido:

```
5
250 1000 50 900 600 1650 650 400 350 1000 1900 100 300 2000 1600
```

Podemos resolverlo a mano para obtener 10 puntos sin programar nada . En el proceso de resolución, iremos encontrando y explicando varias ideas que pueden ser útiles para luego programar un algoritmo de búsqueda exhaustiva que ataque casos más grandes.

Lo primero que podemos hacer para que sea más fácil de analizar, es ordenar los números, ya que su orden no importa, son todas las distancias entre puntos en cualquier orden:

```
50 100 250 300 350 400 600 650 900 1000 1000 1600 1650 1900 2000
```

Vemos que la distancia máxima es 2000, y la distancia máxima siempre se alcanza claramente entre la central y el pueblo más alejado, es decir que ya sabemos $d_5 = 2000$.

Notemos que la distancia d_i para $1 \leq i \leq 4$ debe estar en el conjunto de distancias, pero también debe estar $2000 - d_i$, pues es la distancia del pueblo i al 5. Esto hace que solo algunos valores de los dados sean los posibles d_i, por ejemplo no puede ser $d_i = 50$ porque 1950 no figura entre las distancias.

Así, los únicos valores posibles para los d_i son $100, 350, 400, 1000, 1600, 1650$ y 1900. Notar que el 1000 es posible solamente porque aparece doble, ya que si

apareciera una única vez no sería suficiente, pues al ser $1000 = 2000 - 1000$ tendría que existir otra copia.

Además estos valores están en 4 parejas: el x con el $2000 - x$. Si por ejemplo fuera $d_i = 100$, ya no podría ser que exista un $d_j = 1900$, pues la distancia 1900 correspondería a la distancia del pueblo i al 5. Por lo tanto la solución tiene que consistir sí o sí en elegir un número de cada pareja para obtener los d_1, d_2, d_3, d_4. Pero como las dos copias del 1000 se emparejan entre sí, podemos estar completamente seguros de que uno de los $d_i = 1000$ necesariamente.

Esto ya nos deja únicamente 8 posibilidades para analizar, que podríamos escribir todas y verificar para cada una si se cumplen las distancias o no. Pero podemos razonar un poco más para calcular menos.

Notemos que si en una solución, cambiamos todos los valores d_i por $2000 - d_i$, para $1 \leq i \leq 4$, se obtendrá una nueva solución, porque solamente se reflejó la situación, y entonces el conjunto de distancias no cambia. Esto significa que si hay una solución en la que se elige 1900, en la reflejada se elige 100, o sea que podemos buscar solamente las soluciones donde se elige $d_1 = 100$, y las demás que existan serán sí o sí las reflejadas.

Como ya sabemos entonces que $d_1 = 100$ (tiene que ser el d_1 porque no hay otros candidatos menores en las parejas), para los dos valores de d_i que nos faltan encontrar, deben existir tanto la distancia d_i como la $d_i - 100$, que corresponderán con la distancia del pueblo i a la central y del pueblo i al 1, respectivamente. Y esto sin contar los valores 900 y 1000, pues esos valores ya se corresponden con distancias entre los pueblos que conocemos así que no pueden ser las distancias faltantes.

Pero los únicos valores con esa propiedad son 350 (ya que existe el 250) y 400 (ya que existe el 300) y por lo tanto si hay solución con $d_1 = 100$, la única posibilidad es que sea $d_1 = 100$, $d_2 = 350$, $d_3 = 400$, $d_4 = 1000$ y $d_5 = 2000$. Podemos calcular todas las distancias y corroborar que en efecto estos valores son solución.

Entonces las únicas soluciones que existen son la mencionada y su reflejada, que sería $d_1 = 1000$, $d_2 = 1600$, $d_3 = 1650$, $d_4 = 1900$ y $d_5 = 2000$.

2.2.3.2. Subtarea 2

En esta subtarea, que corresponde al segundo caso de entrada, se nos da una garantía bastante peculiar: tenemos $N = 14$ y sabemos además que existe una solución en la cual **exactamente uno** de los d_i es par.

Esto significa que si incluimos el 0 de la central, para tener los $N + 1$ puntos

de interés entre los cuales conocemos sus $\frac{N(N+1)}{2}$ distancias, hay **exactamente dos puntos** en posición par, y uno de ellos es el 0.

Podemos mirar como antes el máximo de todos los números de la entrada, que resulta ser impar: 1553123251. Como antes, este deberá ser el valor de d_{14} pues la distancia máxima siempre se alcanza desde la central al último pueblo.

Como la diferencia entre par e impar es siempre impar, y la diferencia entre números de igual paridad es siempre par, tendremos que entre las distancias habrá exactamente $2(N-1) = 26$ elementos impares. $N-1$ (es decir, la mitad) de estos elementos corresponderán directamente a los d_i impares, y los otros $N-1$ corresponden a las diferencias $|d_i - p|$ para los distintos d_i impares, siendo $p = d_j$ el único pueblo a distancia par de la central. En particular, uno de estos valores impares será $d_{14} - p$, lo que nos deja solamente 26 candidatos para p, y probamos todos exhaustivamente.

Fijado el valor de p, podemos probar además los $\binom{2(N-1)}{N-1}$ conjuntos de $N-1$ elementos entre los impares, y para cada uno, verificar si es solución, es decir, si es viable que esos $N-1$ elegidos sean los valores d_i impares. Una vez seleccionado un subconjunto, ya tenemos fijados todos los d_i así que podemos calcular las distancias y verificar finalmente si es solución del problema.

Explorando todas estas opciones por fuerza bruta, dependiendo de la implementación se obtiene un tiempo de ejecución aproximado de una hora, lo que podría ser suficiente para resolver el caso en el tiempo de la prueba.

Para que se ejecute más rápido, podemos filtrar y solamente permitir considerar como posibles d_i aquellas distancias impares x para las cuales también exista la distancia $|x - p|$. Agregar esta simple poda reduce el tiempo de ejecución anterior a menos de un segundo. Sería posible realizar más observaciones para reducir aún más el tiempo, pero no es necesario a la hora de resolver este caso en particular.

Si ejecutamos este procedimiento, encontramos una única solución. Esta y su reflejada (que tiene múltiples pueblos pares) son las únicas soluciones que existen para este caso de prueba.

Incluimos a continuación un ejemplo de implementación de estas ideas, que se ejecuta en menos de un segundo:

```
#include <iostream>
#include <set>
#include <vector>
#include <cassert>
```

```
#include <algorithm>

using namespace std;

#define forn(i,n) for(int i=0;i<int(n);i++)
#define dforn(i,n) for(int i=int(n)-1;i>=0;i--)
#define esta(x,c) ((c).find(x) != (c).end())

typedef long long tint;

int main() {
    const int TOT = 105;
    tint distances[TOT] = {999999648,3122844, // ... listado copiado del input
    multiset<tint> alld(distances, distances+TOT);
    vector<tint> impares;
    forn(i,TOT)
    if (distances[i] % 2)
        impares.push_back(distances[i]);
    assert(impares.size() == 26);
    const tint D14 = 1553123251;
    for (tint pdi : impares)
    if (pdi != D14) {
        tint p = D14 - pdi;
        vector<tint> viables;
        for (tint x : impares)
        if (esta(abs(x-p), alld))
            viables.push_back(x);
        forn(mask, 1<<viables.size())
        if (__builtin_popcount(mask) == 13) {
            vector<tint> cand = {0, p};
            forn(i,viables.size())
            if ((mask >> i) & 1)
                cand.push_back(viables[i]);
            multiset<tint> d = alld;
            assert(cand.size() == 15);
            bool failed = false;
            for (int i=0; i<15 && !failed; i++)
            forn(j,i) {
                auto it = d.find(abs(cand[i] - cand[j]));
                if (it == d.end()) {
                    failed = true;
```

```
                break;
            }
            d.erase(it);
        }
        if (!failed) {
            assert(d.empty());
            sort(cand.begin(), cand.end());
            cand.erase(cand.begin()); // Borramos el 0 de la central
            forn(i,cand.size()) {
                if (i > 0) cout << " ";
                cout << cand[i];
            }
            cout << endl;
            // Agregamos la reflejada
            dforn(i, cand.size()-1)
                cout << D14 - cand[i] << " ";
            cout << D14 << endl;
        }
    }
  }
  return 0;
}
```

2.2.3.3. Subtarea 3

La subtarea anterior muy particular nos permite obtener 10 puntos utilizando un enfoque muy específico explicado allí, pero no sirve demasiado para resolver otros casos. Lo que haremos es dar a continuación un algoritmo eficiente de backtracking (http://wiki.oia.unsam.edu.ar/algoritmos-oia/backtracking) para el problema, que obtiene 80 puntos resolviendo automáticamente los dos casos anteriores y todos los de la subtarea 3. Este algoritmo de backtracking es explicado por Lemke, Skiena y Smith en su paper *Reconstructing Sets From Interpoint Distances (2003)*. Si bien el paper estudia muchos otros resultados y teoremas avanzados relacionados al problema de las distancias, el algoritmo de backtracking en sí es simple y de hecho fue desarrollado durante la prueba por algunos participantes.

La idea fundamental consiste en ir descubriendo los puntos **desde los extremos hacia el interior**. Además, durante la búsqueda por backtracking, se van tachando las distancias del listado ya utilizadas.

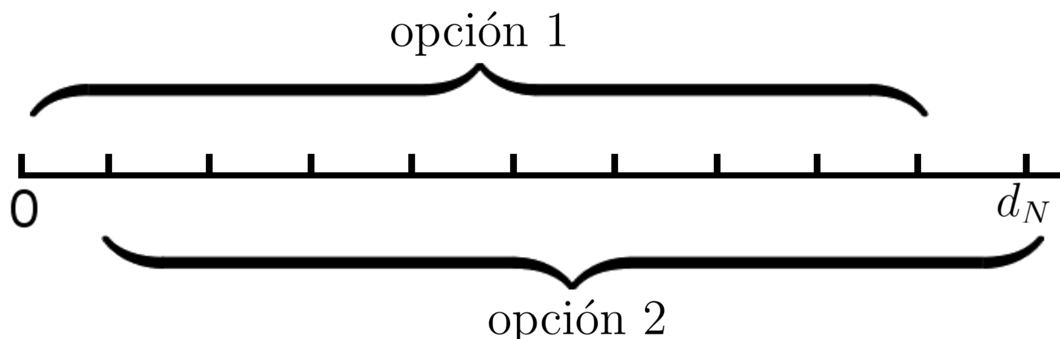
Primero que nada, podemos ubicar los puntos 0 (correspondiente a la central) y d_N (que se obtiene simplemente tomando la distancia más grande que aparece

en el archivo de entrada). Con ellos fijados, quedan $N-1$ puntos adicionales por descubrir, y nos quedan $\frac{N(N+1)}{2}-1$ distancias por utilizar (pues de todas las posibles, únicamente tenemos ya realizada la máxima, entre el punto 0 y el N).

Cuando hayamos ubicado ya K puntos de los $N+1$ durante algún paso de ejecución del backtracking, tendremos ya tachadas $\binom{K}{2} = \frac{K(K-1)}{2}$ distancias del listado total de $\frac{N(N+1)}{2}$: aquellas que correspondan a todo par de puntos ya ubicados. Por eso inicialmente arrancamos con $K=2$ (los puntos extremos ya ubicados) y exactamente una distancia del listado tachada.

La observación clave es a partir de ahora, en cada paso, considerar **la máxima** distancia aún no tachada. Necesariamente, esta distancia tiene que ser una de únicamente dos opciones posibles:

1. La distancia desde el 0 hasta el punto aún por descubrir que esté más alejado de la central - es decir, el siguiente por la derecha.
2. La distancia desde el pueblo N hasta el punto aún por descubrir más cercano a la central - es decir, el siguiente por la izquierda.



Esto es porque cualquier otra distancia aún no tachada, como debe involucrar necesariamente a algún punto aún por descubrir, corresponde a un intervalo de la recta que estará completamente contenido en alguno de estos dos, y por lo tanto no puede ser más largo. Como no tenemos ningún criterio que nos indique cuál de estos dos casos tomar como correcto, lo que hacemos es probar ambos sucesivamente, y así obtenemos una implementación de backtracking recursiva.

Si D es la siguiente distancia más grande, cuando elegimos la opción 1 el siguiente punto se ubicará directamente en D. Y cuando elegimos la opción 2, el siguiente punto se ubicará en $d_N - D$. Una vez que fijamos la ubicación de este nuevo punto, nos genera K distancias nuevas descubiertas: una con cada punto que ya teníamos ubicado. Lo que hacemos es verificar que todos los números que así

se obtienen sean distancias aún no tachadas, y las tachamos. Si para alguna de estas distancias no podemos encontrar el número correspondiente, significa que la elección es imposible y ya podemos podar la búsqueda y no considerar esa opción. El algoritmo encuentra una solución cada vez que logra tachar todas las distancias (necesariamente, tras haber colocado el último punto adicional a los extremos, es decir, el $N-1$-ésimo).

Si guardamos en un `set` las distancias restantes, tendremos disponibles las operaciones de agregar, sacar y consultar si una distancia existe, todas en tiempo $O(\lg N)$. En cada paso se realizarán $O(N)$ de estas operaciones. La complejidad de este algoritmo resulta ser entonces $O(2^N N \lg N)$, ya que cada paso se procesa en $O(N \lg N)$, y el árbol de posibilidades tiene profundidad $N-1$ y en cada paso a lo sumo se bifurca en dos posibilidades.

No obstante, en la práctica funciona muchísimo más rápido que lo que esta complejidad sugiere para casi todo caso de entrada, pues este análisis de complejidad se obtiene suponiendo que en muchos pasos ocurre que ambos caminos resultan consistentes, pero es muy difícil que ninguna distancia falle rápidamente en casos reales tras haber realizado una elección incorrecta que no conduce a ninguna solución.

Los casos de prueba incluyen casos difíciles construidos para exhibir este tiempo exponencial del algoritmo, pero igualmente la constante empírica es baja, ya que no ocurre que en todos los $N-1$ pasos haya dos opciones viables.

2.2.3.4. Subtarea 4

En esta subtarea tenemos valores grandes de $N = 100$, y el algoritmo exponencial anterior no terminará en tiempo razonable. Debemos implementar una solución por backtracking, pero que realice la búsqueda de manera diferente.

La clave es que en esta subtarea se nos garantiza que todos los $d_i \leq 106$, es decir, las coordenadas de los pueblos toman valores muy pequeños. En particular, como son todas diferentes, tenemos 100 pueblos que repartir entre un máximo de 106 ubicaciones posibles (los enteros desde 1 hasta 106). Esto da un total máximo posible de $\binom{106}{100} = \binom{106}{6} = 1705904746 \approx 1{,}7 \cdot 10^9$. Este número es bastante grande, pero es un número de posibilidades que las computadoras actuales pueden procesar en minutos, mientras que no hay ninguna esperanza de realizar 2^{100} operaciones durante el tiempo de prueba [4].

[4]Necesitaríamos computadoras aproximadamente mil veces más rápidas que las actuales para que alcance la edad del universo.

Lo que podemos hacer es ir posición por posición, desde la 1 hasta la 106, y en cada paso, elegimos si ubicar allí un pueblo o no. Cuando ubicamos pueblos, tachamos las distancias utilizadas por los que ya tenemos, tal cual como hacíamos antes, podando cuando no encontramos las distancias que deberíamos encontrar.

Vale la pena mencionar que como las distancias en este caso son números desde 1 hasta 106, en lugar de un `set` podemos utilizar un simple arreglo global de enteros, que guarde la cantidad de copias de cada distancia posible que existen. De esa forma, agregar y sacar es sumar uno y restar uno al contador adecuado, que es una operación extremadamente eficiente. Este algoritmo de backtracking es más que suficientemente rápido para resolver estos casos, pues a lo sumo explorará las $\binom{106}{100}$ posibles distribuciones, lo cual debería poder realizarse exhaustivamente en aproximadamente algunos minutos o una hora, según la implementación: pero de hecho gracias a las podas mencionadas, se explorará mucho menos y la ejecución completa terminaría en algunos segundos. Este backtracking alternativo permite obtener los 20 puntos de los últimos dos casos, pero no sirve para resolver los anteriores, pues se basa fuertemente en que las distancias son pequeñas.

Capítulo 3

Certamen Jurisdiccional

3.1. Nivel 1

3.1.1. Problema 1: Programando calculadoras [calculadora]

http://juez.oia.unsam.edu.ar/#/task/calculadora/statement

Este era el problema más simple en el certamen jurisdiccional OIA 2019. En el enunciado se describen con precisión 4 funciones de la calculadora que se deben implementar: lo único que hay que hacer es codificarlas en un lenguaje de programación para obtener el puntaje. Cada una de las 4 funciones puede implementarse por separado, para obtener 25 puntos por cada una, y 100 puntos por un programa que implemente las 4 correctamente.

Algunos comentarios sobre la codificación en C++ o Java:

- Para retornar un resultado de una función, utilizamos la instrucción `return`. Con ella se indica cuál es el valor solución que debemos retornar en la función que estemos codificando.
- La suma y el producto se indican con los símbolos `+` y `*` respectivamente.
- Para obtener la paridad y decidir si un número es par o impar, se puede utilizar el operador `%`, que calcula el resto en la división. De esa manera, para x no negativo tenemos que `x%2` da por resultado 1 si x es impar, y 0 si x es par.
- Para implementar la operación de selector, que es la más difícil del ejercicio, utilizaremos la instrucción `if` / `else`. Esta permite verificar si se cumple una condición entre paréntesis, y de ser así, nos permite dividir la ejecución en dos

caminos, según si esa condición se cumple o no. Para la operación selector esto es necesario pues debemos seleccionar qué operación realizar dependiendo del valor de una variable, algo que no podemos saber previamente antes de que el programa se esté ejecutando, así que hay que escribir un programa que maneje correctamente ambos casos.

- Podemos aprovechar funciones ya escritas, sin necesidad de copiar su código interno muchas veces. Para esto, basta con escribir el nombre de la función y a continuación sus parámetros (valores que le damos a esa función como datos para trabajar) entre paréntesis.

A continuación mostramos una implementación en C++ de las funciones, que obtendría 100 puntos:

```
int suma(int a, int b) {
    return a+b;
}

int producto(int a, int b) {
    return a*b;
}

int paridadSuma(int a, int b) {
    return (a+b)%2;
}

int selector(int op, int a, int b) {
    if (op == 0) return suma(a,b);
    else if (op == 1) return producto(a,b);
    else return paridadSuma(a,b);
}
```

3.1.2. Problema 2: Organizando el Librero [librero1]

http://juez.oia.unsam.edu.ar/#/task/librero1/statement

La solución a este problema se explica en la versión de nivel 3.

Ver librero3 en (3.3.2).

3.1.3. Problema 3: Preparando la Receta [receta]

http://juez.oia.unsam.edu.ar/#/task/receta/statement

En este problema, recibimos dos arreglos: uno con los ingredientes que hay en la heladera, y otro los ingredientes necesarios. Hay que calcular cuántos ingredientes faltan en la heladera para poder cocinar la receta. Ambas listas pueden tener repetidos, pero si los hay debemos contar una sola vez cada uno.

Podemos utilizar fors y sumar a un contador: vamos recorriendo los ingredientes de la receta uno a uno, con un primer for. Verificamos con un segundo for que no haya aparecido antes, para evitar repetidos. Si ya apareció, lo ignoraremos. Pero si no apareció, y es un ingrediente nuevo, hay que verificar si está en la heladera o no. Si no está en la heladera hay que contarlo como un ingrediente faltante, para lo cual podemos ir llevando una variable contadora.

De esta forma, tenemos una solución de 50 puntos (calcula correctamente la cantidad de ingredientes faltantes):

```
int receta( vector<string> heladera,
            vector<string> ingredientes,
            vector<string> &faltantes) {
    int cantidadFaltantes = 0;
    for (int i=0; i<int(ingredientes.size()); i++)
    {
        bool visto = false;
        for (int j=0; j<i; j++)
        if (ingredientes[i] == ingredientes[j])
            visto = true;
        if (!visto)
        {
            bool presente = false;
            for (int j=0; j<int(heladera.size()); j++)
            if (heladera[j] == ingredientes[i])
                presente = true;
            if (!presente)
                cantidadFaltantes++;
        }
    }
    return cantidadFaltantes;
}
```

¿Qué nos falta para obtener 100 puntos con la solución anterior? Calcular no solamente la cantidad, sino también cuáles son los ingredientes que faltan, y guardar eso en el arreglo de faltantes. Siempre conviene limpiar primero el arreglo si vamos a llenarlo más tarde con datos, como precaución por si no estuviera vacío inicialmente.

Esto es un cambio sencillo al código anterior ya que por cómo lo armamos, fuimos ingrediente por ingrediente y descubrimos cuáles son los que no están: solo hay que cambiar el `cantidadFaltantes++` por un `push_back` que agregue el ingrediente a la respuesta. El código final queda:

```
int receta( vector<string> heladera,
            vector<string> ingredientes,
            vector<string> &faltantes) {
    faltantes.clear();
    for (int i=0; i<int(ingredientes.size()); i++)
    {
        bool visto = false;
        for (int j=0; j<i; j++)
        if (ingredientes[i] == ingredientes[j])
            visto = true;
        if (!visto)
        {
            bool presente = false;
            for (int j=0; j<int(heladera.size()); j++)
            if (heladera[j] == ingredientes[i])
                presente = true;
            if (!presente)
                faltantes.push_back(ingredientes[i]);
        }
    }
    return int(faltantes.size());
}
```

Alternativamente, se puede utilizar la estructura de datos set, que ya viene programada y permite eficiente y cómodamente almacenar elementos sin repeticiones, contar, y consultar si un elemento está o no. Dejamos a continuación una solución de ejemplo. Se puede leer sobre set en http://wiki.oia.unsam.edu.ar/cpp-avanzado/set

```
// Macro util para saber si un elemento x esta en un set/map c
```

```
#define esta(x,c) ((c).find(x) != (c).end())

int receta( vector<string> heladera,
            vector<string> ingredientes,
            vector<string> &faltantes) {
    // Eliminamos repetidos en ambos conjuntos usando la STL
    set<string> heladera_sin_repetidos;
    for (const auto &ingrediente : heladera)
        heladera_sin_repetidos.insert(ingrediente);
    set<string> ingredientes_sin_repetidos;
    for (const auto &ingrediente : ingredientes)
        ingredientes_sin_repetidos.insert(ingrediente);

    // Calculamos los ingredientes necesarios que no estan en la heladera
    faltantes = {};
    for (const auto &ingrediente : ingredientes_sin_repetidos)
        if (!esta(ingrediente, heladera_sin_repetidos))
            faltantes.push_back(ingrediente);

    return int(faltantes.size());
}
```

3.1.4. Problema 4: Enfrentando Monstruos [batamon]

http://juez.oia.unsam.edu.ar/#/task/batamon/statement

Esta explicación incluye:

1. Ideas detalladas que llevan a un algoritmo muy eficiente y correcto que nos dice a quién enfrentar con quién.
2. Pequeña explicación de por qué el algoritmo anda rápido.
3. Qué se podía haber hecho para resolver algunas subtareas si la solución completa no se nos ocurría.
4. Cómo programar las ideas mencionadas en 1.
5. Un código escrito en C++ que da 100 puntos.

3.1.4.1. Ideas

Iremos desglosando este problema de a poco.

¿Qué pasa si la hechicera ve que uno de sus monstruos es muy débil? Por ejemplo, supongamos que el de menor nivel de poder fuera más débil que todos los demás monstruos existentes. En ese caso, como sabe que lamentablemente este monstruo va a perder siempre, podría decidir enfrentarlo contra el más fuerte del rival, para que de esa manera el rival tenga que gastar su mejor monstruo en una batalla que la hechicera perdería de cualquier manera.

¿Y qué pasa si ve que uno de sus monstruos es el más poderoso de todos? En ese caso, le convendría enfrentarlo con el más fuerte de su rival, ya que al poder ganar **cualquier** batalla, este monstruo tendrá el efecto de borrar a uno de los del rival a elección, con idéntico efecto en la cantidad de batallas ganadas sin importar cuál se borre.

¿Y qué pasa si ve que su rival tiene el monstruo más fuerte de todos? Algo parecido a lo que ocurría en el caso de tener el más débil de todos: Lo ideal es enfrentar al más débil de los monstruos de la hechicera contra el monstruo más fuerte rival, para que justamente su rival desperdicie todo este poder del monstruo más poderoso: como sí o sí este monstruo luchará y vencerá uno de los de la hechicera, borrándolo, conviene elegir que borre el más débil.

Pensando en esto, podemos ya pensar la estrategia de la hechicera: Si el más fuerte que tiene es más fuerte (o igual porque con su poder puede inclinar la balanza para ganar si hay empate) que el más fuerte de su rival, los enfrenta y gana esta batalla, sacándole el más fuerte. Y si no, o sea si el más fuerte de ella es más débil que el más fuerte de su rival, agarra al suyo más débil y lo enfrenta contra este más fuerte de todos, ya que sabemos que cualquiera suyo iba a perder contra este, y así logra que el que pierda sea el menos poderoso y conserva los demás.

¿Y qué ocurre una vez pasada esa batalla? Ocurre que, sin importar el resultado de la primera batalla, la situación a continuación es la misma que habría habido inicialmente si tuvieran $N - 1$ monstruos cada uno, y hubiera que plantear la estrategia ideal para enfrentar esos monstruos. Por lo tanto, como la situación es análoga a la que teníamos al comienzo (pero con un monstruo menos en cada bando) podemos repetir el mismo razonamiento y seguir realizando esos pasos hasta que ya no queden monstruos por enfrentar.

Notar que esto no pasaría si el enunciado del problema fuera diferente: si de alguna forma se hubiera podido reutilizar en batallas posteriores la diferencia de poder entre los monstruos, de modo que al vencer con uno de poder 10 a uno de poder 4 se pudiera almacenar y reutilizar más tarde ese excedente, la situación luego de cada paso sería diferente y no podríamos volver a aplicar el razonamiento del comienzo sin cambios, pues sería una situación diferente donde tenemos excedente

del paso anterior. Pero en este problema, **La situación en cada paso no depende para nada de cómo se llegó hasta allí**. Lo único que nos importa después de cada batalla, es qué monstruos quedan.

Veamos cómo implementar esta solución. Cuando la hechicera tiene que armar una batalla:

- O bien pone a su mejor monstruo contra el mejor de su rival (si de esta manera gana).
- O bien pone a su peor monstruo contra el mejor de su rival (si el mejor de su rival es mejor que todos los de la hechicera).

Veamos que **siempre estamos usando o bien al peor, o bien al mejor**. Esto sugiere muy fuertemente **ordenar a los monstruos** de menor a mayor, porque entonces el mejor sería el último, y el peor el primero. Y entonces en cada paso siempre estaríamos sacando el último monstruo del rival, y el primero o el último de la hechicera.

Supongamos por ejemplo que la hechicera utilice su mejor monstruo. Para la siguiente batalla de vuelta se deberá saber quiénes son su peor y mejor monstruo. El peor es el primero, igual que antes. Y el mejor resulta ser el penúltimo, ya que el último, que era el mejor, ya batalló. Si después elige de vuelta a su mejor monstruo, el mejor restante va a ser el antepenúltimo. Y así sucesivamente, siempre que use al mejor monstruo, el mejor entre los que no utilizó se va a desplazar una posición hacia la izquierda, y siempre que use a su peor monstruo el peor entre los que no utilizó se desplazará una posición hacia la derecha. Notemos que según nuestra estrategia, en cada turno necesitamos únicamente conocer quién es el mejor y el peor.

Entonces, para programar esto, podemos tener dos variables, una que nos indique cuál es el peor de los monstruos que le quedan a la hechicera, y la otra cuál es el mejor. Si indexamos arreglos desde 0, la variable que indica cuál es el peor, va a empezar en 0, porque al ordenarlos el peor queda al principio, y la variable que indica el mejor va a ser $N-1$. Llamémoslas para este análisis A y B respectivamente.

¿Cómo sería el procedimiento? Se considera el mejor monstruo del rival de la hechicera. Si su poder es más grande que el del mejor de ella, se enfrenta contra el peor, así que se incrementa A en uno para moverse hacia la derecha. Si no, es decir si el más grande del rival es peor o igual que el mejor de la hechicera, se enfrentan entre sí, y entonces se decrementa B en uno. Además, en este último caso se debe sumar uno a la cantidad de batallas ganadas.

3.1.4.2. Cuánto tarda

Visto cómo resolver el problema completo, pensemos qué hay que hacer. Primero, se deben ordenar los monstruos de menor a mayor (tanto los de la hechicera como los de su rival). ¿Y después? Se tienen dos variables, una comienza a la izquierda y la otra a la derecha. En cada uno de los N pasos correspondientes a una batalla, Lo único que hacemos es comparar el monstruo de la posición de la derecha de la hechicera con el mejor que queda de su rival. Si la hechicera puede ganar, suma 1 a la cantidad de victorias y resta 1 a la variable de la derecha, y si no simplemente suma 1 a la de la izquierda. Además, hay una variable que nos va diciendo quién es el mejor del rival, que como nuestra idea era siempre agarrar a su mejor monstruo, simplemente empieza a la derecha de todo y va restando de a uno.

Entonces, por cada batalla simplemente sumamos y/o restamos uno a un par de variables.

Pero no nos olvidemos del primer ordenamiento que hacemos.

Entonces, hablando en notación asintótica[1], el algoritmo tiene una complejidad temporal $O(N + Nlog(N)) = O(Nlog(N))$.

3.1.4.3. Subtareas

Prestar a atención a las subtareas es muy importante.

Supongamos que nos aseguran que "cada persona tiene 2 monstruos". Podemos ahorrarnos todo el trabajo de ordenarlos y pensar en un algoritmo complicado y general como el anterior. En su lugar podemos razonar simplemente que hay únicamente dos opciones: o bien se enfrentan el primero con el primero y el segundo con el segundo, o bien se cruzan. Entonces, podemos evaluar la cantidad de victorias que ocurren en cada una de estas dos posibilidades, y elegir el caso más favorable, es decir, tomar el máximo de las victorias. Esto es así porque la hechicera enfrenta a los monstruos a su conveniencia, así que al haber solamente 2 monstruos, podemos explorar todas (las únicas dos) posibilidades.

Esta era la primera subtarea del problema, que indica $N = 2$. Un programa que la resuelve es muchísimo más sencillo de obtener que una solución completa de 100 puntos, pero ya permite obtener 5 puntos.

Esto es algo muy interesante de la programación: Muchas veces no vamos a estar en una situación tan sencilla, en la que tengamos tan pocas opciones que analizar. ¡Pero si las tenemos, lo podemos aprovechar!

[1] http://wiki.oia.unsam.edu.ar/algoritmos-oia/complejidad

La solución anterior más compleja funciona también cuando $N = 2$, por supuesto, pero lleva probablemente mucho más tiempo de pensar y programar. Si por ejemplo supiéramos que siempre $N = 2$ en todas las subtareas, en vez de sólo en la de 5 puntos, la mejor estrategia sería programar esta solución simple y listo, incluso si somos capaces de pensar y programar la otra solución más general, simplemente para ahorrar tiempo y reducir la posibilidad de equivocarnos.

¿Y si $N \leq 9$? En este caso ya no resulta práctico analizar manualmente todas las posibilidades de cruce de monstruos[2], pero lo interesante es que sí podemos hacer que la computadora analice todos los ordenamientos posibles por nosotros.

Es necesario hacerlo de manera eficiente: No vamos a crear un programa que analice para cada ordenamiento de los monstruos de la hechicera, cada ordenamiento de los de su rival. Esto no hace falta porque mandar a batallar a los monstruos de la hechicera en el orden $2, 1, 3$ y los de su rival en el orden $3, 1, 2$, genera los mismos enfrentamientos que mandar a los de la hechicera en el orden $2, 3, 1$ y los de su rival $3, 2, 1$, por ejemplo.

En general, siempre podemos dejar "quietos" los monstruos de su rival, y probar cada ordenamiento de los monstruos de la hechicera: para cada ordenamiento posible, contaremos cuántas veces gana, y nos quedaremos con la mejor. Podemos implementar esta solución en pocas líneas utilizando la técnica de fuerza bruta[3], y una función no muy utilizada pero muy práctica, llamada en C++ `next_permutation`[4].

Por último, también conviene analizar las cotas no sólo de N sino de H y R. Una de las subtareas especifica que $R \leq 3, H \leq 3$. Es decir, todos los poderes serán 1, 2 o 3.

Podemos pensar una estrategia para este caso sencillo: Todos los 3 de la hechicera los pongo contra los 3 de su rival. Si a la hechicera le quedan monstruos de poder 3 y a su rival no, ahora es lo mismo que los 3 sean 2, total con eso ya le ganamos al mejor de su rival. Y de vuelta, juntamos a los 2 y 3 de la hechicera con los 2 de su rival hasta agotar los de algún lado. Y después pasamos a los 1 de la hechicera, que sólo le van a ganar a los 1 de su rival.

Y programar esto lo podemos hacer muy fácil. Podemos tener 6 contadores: h_1, h_2, h_3 nos dirán la cantidad de 1, 2 y 3 en los monstruos de la hechicera, y r_1, r_2, r_3, equivalente para su rival. Luego sabemos que gana $min(h_3, r_3)$. Después, si $h_3 > r_3$, realizamos la asignación $h_2 \leftarrow h_2 + h_3 - r_3$, los que sobran que ahora

[2] Puesto que hay $9! = 362880$ posibilidades diferentes.
[3] http://wiki.oia.unsam.edu.ar/algoritmos-oia/backtracking
[4] http://wiki.oia.unsam.edu.ar/cpp-avanzado/algorithm/next-permutation

van a batallar contra los monstruos de 2 de su rival. Ahora, con los monstruos que nos quedan vamos a ganar $min(h_2, r_2)$. Y de vuelta si nos sobran asignamos $h_1 \leftarrow h_1 + h_2 - r_2$. Por último, vamos a ganar $min(h_1, r_1)$. Es más complicado lograr imprimir los índices de los monstruos, pero con esta estrategia mucho más sencilla que no utiliza fors salvo para contar los valores iniciales de los contadores y luego hace un par de cuentas, al menos obtenemos el 70 % del puntaje.

3.1.4.4. Cómo escribir el código

Volvemos a la solución general dada anteriormente. Para contar la cantidad de victorias máxima no hay mucho detalle: Ordenamos, inicializamos las variables que mencionamos, y vamos viendo batalla por batalla si la hechicera usará a su más débil o su más fuerte contra el más fuerte de su rival.

Lo más complicado es conseguir el 30 % del puntaje que corresponde a lo que se conoce como "reconstruir" la solución. No debemos dar solamente cuál es el valor del mejor puntaje, sino **también cómo obtenerlo**.

Es un muy buen ejercicio pensar cómo se podría hacer esto antes de leerlo del párrafo siguiente, una vez que ya entendimos la solución para calcular la cantidad de victorias.

La respuesta que tenemos que dar son índices de cómo estaban los monstruos **al principio, antes de ordenar**. Un truco muy usado es en vez de tener solamente un arreglo con los poderes de los monstruos, tener un arreglo con un par de números, el poder y el índice del monstruo. Así, por más que los cambiemos de lugar sabemos en dónde estaban al principio y podremos dar esta respuesta. La forma más conveniente de combinar y tener juntos los dos valores de poder e índice inicial es crear nuestro propio tipo `Monstruo`.

Teniendo esta idea, no hay que hacer mucho más: Cuando comparamos al monstruo de la hechicera de valor h_V y posición inicial h_p que sería el más poderoso que le queda, contra el monstruo de su rival de valor r_V y posición inicial r_p, si los queremos enfrentar (porque la hechicera gana), guardamos en algún otro arreglo respuesta el valor h_p en la posición r_p de este arreglo, para decir que se enfrentarán estos monstruos. Si la hechicera pierde, guardamos la posición inicial de su monstruo más débil (que tenemos a la izquierda de todo).

Luego, al terminar, tendremos en cada posición de este nuevo arreglo, qué monstruo de la hechicera se enfrenta con cada uno de estos, y simplemente imprimimos este arreglo de izquierda a derecha.

3.1.4.5. Código en C++

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

#define all(c) (c).begin(), (c).end()

int batamon(vector<int> hechicera,
            vector<int> rival,
            vector<int> & enfrentamientos) {

    int n = int(hechicera.size());
    struct Monstruo {
        int poder, indice;

        bool operator <(const Monstruo &otro) const {
            return poder < otro.poder;
        }
    };
    vector<Monstruo> hechiceraMonstruos, rivalMonstruos;
    for(int i = 0; i < n; i++) {
        hechiceraMonstruos.push_back({hechicera[i], i});
        rivalMonstruos.push_back({rival[i], i});
    }
    sort(all(hechiceraMonstruos));
    sort(all(rivalMonstruos));
    enfrentamientos.clear();
    for(int i=0; i<n; i++)
        enfrentamientos.push_back(-1);
    int masDebilHechicera = 0;
    int masPoderosoHechicera = n-1;
    int ganaHechicera = 0;
    for (int masPoderosoRival = n-1; masPoderosoRival >= 0; masPoderosoRival--) {
        if (hechiceraMonstruos[masPoderosoHechicera].poder >=
            rivalMonstruos[masPoderosoRival].poder) {
            ganaHechicera++;
```

```
            enfrentamientos[rivalMonstruos[masPoderosoRival].indice] =
                    hechiceraMonstruos[masPoderosoHechicera].indice;
            masPoderosoHechicera--;
        } else {
            enfrentamientos[rivalMonstruos[masPoderosoRival].indice] =
                    hechiceraMonstruos[masDebilHechicera].indice;
            masDebilHechicera++;
        }
    }
    // sumamos uno para que los indices empiecen en 1 en vez de 0
    for(int i=0; i<n; i++)
        enfrentamientos[i]++;
    return ganaHechicera;
}

// EVALUADOR LOCAL.
// EN LA PRUEBA YA VIENE PROGRAMADO.
// SE INCLUYE AQUI POR COMODIDAD PARA EL LECTOR.

int main() {
    int n;
    cin>>n;
    vector<int> hechicera(n), rival(n);
    for(int i = 0; i < n; i++)
        cin>>hechicera[i];
    for(int i = 0; i < n; i++)
     cin>>rival[i];
    vector<int> enfrentamientos;
    int ganaHechicera = batamon(hechicera, rival, enfrentamientos);
    cout<<ganaHechicera<<endl;
    for(int i = 0; i < n; i++) {
        if (i > 0)
            cout << " ";
     cout<<enfrentamientos[i];
    }
    cout << endl;
}
```

3.2. Nivel 2

3.2.1. Problema 1: Tetris de Plástico [plastetris]

http://juez.oia.unsam.edu.ar/#/task/plastetris/statement

Este problema tiene algo muy lindo que vamos a ir descubriendo a lo largo de esta explicación.

Primero, algo que es sencillo pero súper útil es notar que necesitamos que haya exactamente cuatro cruces (`'X'`) en todo el tablero. Si hay más, o hay menos, la respuesta será NO.

Ahora, una vez que sabemos que hay 4 cruces, algo que se me ocurre que podemos hacer es una función para verificar si en el tablero tenemos cada una de las piezas. Entonces por ejemplo una función que nos diga si en tablero las cruces están todas en la misma fila o columna; otra función para que nos diga si forman un cuadrado; otra para que nos diga si forman una T, con 3 cruces y una adyacente a la cruz del medio de esas 3; y otra función para la forma que parece la más complicada, que va como en escalerita.

Algo súper útil de pensar por separado cada figura, es que si no nos saliera formar la 'escalerita', haciendo solamente las figuras de 'palito' y de 'cuadrado', ya tenemos 21 puntos por las subtareas, que son importantes.

Si de una quisiéramos ir por el problema completo, sin ninguna lógica en particular como hacer pieza por pieza, podemos simplemente pensar todas las formas de ubicar las figuras y hacer un 'IF' por cada una de ellas: Para el palito hay 8, 4 filas y 4 columnas; para el cuadrado hay 9, la esquina de arriba izquierda puede estar en cualquiera de las 3 filas de arriba y de las 3 columnas de la izquierda; y así siguiendo, no serían muchas posibilidades, pero hay que prestar atención a no olvidarse ninguna.

En todo caso, si empezamos con el cuadrado y con el palito, si nos aseguramos de tener esas 17 configuraciones, sabemos que los 21 puntos de las subtareas también los tendremos.

Algo que tiene bueno darse cuenta de que hay tan pocos casos es que cuando programamos esto, podemos probar como entrada cada uno de ellos sin que nos tome muchos minutos y asegurarnos este puntaje.

Comentario sobre esto: Hay algo muy divertido e interesante que se conoce como 'testear', que podemos hacerlo, como mencioné recién, a mano, probando cada uno de los 17 casos; pero como capaz vamos a seguir cambiando el código, y

queremos asegurarnos de que nuestro programa responda bien, podemos escribir en el mismo código, por ejemplo en una función aparte, una función de testeo, que tenga un tablero (o sea, cuatro strings) y la respuesta deseada, ya sea SI o NO. Desde ahí, llamamos a nuestra función 'plastetris(cajita)' y comparamos lo que devuelve con la respuesta deseada que escribimos a mano.

Ahora sí, algo interesante: De cualquier manera que pongamos 4 cruces, siempre y cuando desde cada cruz se pueda llegar a las demás yendo por casillas adyacentes cruces (no vale que se toquen en una esquina), ¡vamos a formar una figura de tetris!
Es decir, no vale esto, pues no estarían conectadas:

.	X	.	.
.	X	.	.
.	.	X	X
.	.	.	.

Pueden probar, poniendo cuatro cruces de cualquier manera que se cumpla esto, y van a ver que no hay otra forma. De hecho, así se forman las 'piezas de tetris': Son todas las figuras que se formen con 4 cuadraditos y estén todos conectados.
Entonces, si nos damos cuenta de esto, podría ser más fácil de resolver: Simplemente hay que ver que haya 4 cruces, y que estén todas conectadas entre sí.
Para hacer eso, podemos usar un algoritmo que se llama BFS y tiene algo de teoría de lo que se llaman grafos. De ese algoritmo, y de grafos, pueden leer en la wiki, concretamente en: `http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos` y en `http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/bfs`.

Pero como este tablero es chiquito, podemos pensar cómo hacer, por más que no sea tan 'rápido' o eficiente, para chequear esta condición en este tablero.
Pensemos que si tenemos 4 cruces, podemos tener:

- Cuatro grupitos de 1 cruz.
- Dos grupitos de 1 cruz y un grupo de 2
- Un grupo de 1 cruz y uno de 3
- Dos grupos de 2

- Un grupo de 4

Entonces, básicamente, para saber si estamos en el último caso que es el que nos diría si las cruces forman una pieza de tetris, podemos buscar los otros: Si hay un grupo de 1 sola cruz, estamos en alguno de los primeros tres casos; si no hay un grupo de 1 y hay un grupo de exactamente 2 cruces, estamos en el penúltimo caso.

3.2.1.1. Truco para hacer el código más legible

Algo que nos puede ayudar para esto, que es un truco muy usado, es decir 'en vez de que tablero termine ahí, me imagino que le agrego arriba, abajo, a la izquierda y la derecha, todos grupos de puntitos. Esto es que si, por ejemplo, estamos viendo si una cruz está aislada, no tener que ver si justo está en el borde o no, simplemente ponemos puntitos (que tiene sentido porque son como casillas vacías, o sea si una cruz está en la primera fila es como que arriba no tenga cruces vecinas, que es lo que vamos a chequear).

3.2.1.2. Código

Dejo un código acá que hace eso: Recorre el tablero, y cuando encuentra una cruz, se fija que arriba, abajo, a la derecha y a la izquierda no haya nadie. Si encuentra esto, es que hay un grupito de una sola cruz, entonces devuelve 'NO'. Y si no lo encuentra, veamos que si hay dos grupitos de 2, todas las cruces tienen una sola cruz vecina. Mientras que en cualquier figura válida hay por lo menos una cruz con más de 1 cruz adyacente. Entonces me fijo esto, que si todas tienen una sola cruz vecina tengo dos grupitos de 2, y si no, significa que tengo una figura válida.

```
string plastetris(vector<string> cajita) {
    vector<string> extendido;
    extendido.push_back("......");
    for(int i=0; i<4; i++){
        extendido.push_back("." + cajita[i] + ".");
    }
    extendido.push_back("......");
    int cantidadDeCruces = 0;
    bool cruzAislada = false;
    bool todasTienen1vecina = true;
```

```
    for(int i=1; i<5; i++){
        for(int j=1; j<5; j++){
            if(extendido[i][j] == 'X'){
                cantidadDeCruces++;
                int vecinas = 0;
                if(extendido[i-1][j] == 'X')vecinas++;
                if(extendido[i+1][j] == 'X')vecinas++;
                if(extendido[i][j-1] == 'X')vecinas++;
                if(extendido[i][j+1] == 'X')vecinas++;
                if (vecinas > 1){
                    todasTienen1vecina = false;
                }
                if (vecinas == 0){
                    cruzAislada = true;
                }
            }
        }
    }
    if(cantidadDeCruces != 4 || cruzAislada || todasTienen1vecina){
        return "NO";
    }
    return "SI";
}

// EVALUADOR LOCAL
// EN LA PRUEBA YA VIENE PROGRAMADO
// ESTA INCLUIDO PARA QUE PUEDAN TESTEAR EN SUS COMPUS
int main() {
    vector<string> cajita;
    for(int i=0; i<4; i++){
        string fila;
        cin>>fila;
        cajita.push_back(fila);
    }
    cout<<plastetris(cajita)<<endl;
}
```

3.2.2. Problema 2: Preparando Recetas [recetas]

http://juez.oia.unsam.edu.ar/#/task/recetas/statement

Este problema es una versión un poquito más difícil del problema `receta` de nivel 1 (3.1.3). Recomendamos leer la solución a ese problema primero.

Teniendo solucionado el problema `receta`, lo que observamos es que en la versión de nivel 2 se tienen varias recetas diferentes, en lugar de una sola. Podemos saber cuáles recetas son posibles de realizar, utilizando un código similar al de nivel 1: De hecho para nivel 1 pudimos calcular con precisión cuántos ingredientes faltaban, así que usando esa misma técnica, cuando la cantidad de faltantes sea 0 es que la receta puede realizarse.

Podemos por lo tanto primero que nada iterar utilizando un for todas las páginas del libro de recetas, y por cada una utilizar a su vez la misma técnica con sets que se explicó al final de la solución nivel 1, para saber eficientemente si la receta puede realizarse o no. Para aquellas recetas que puedan realizarse, hacemos `push_back` a un arreglo resultado que indique justamente las páginas de las recetas posibles.

A continuación se transcribe el código de una solución de 100 puntos:

```
// Macro util para saber si un elemento x esta en un set/map c
#define esta(x,c) ((c).find(x) != (c).end())

int recetas(vector<string> heladera,
            vector<vector<string> > libro,
            vector<int> &realizables) {
    // Eliminamos repetidos usando la STL
    set<string> heladera_sin_repetidos;
    for (const auto &ingrediente : heladera)
        heladera_sin_repetidos.insert(ingrediente);

    // Para cada receta verificamos la inclusion
    realizables = {};
    for(int pagina = 0; pagina < int(libro.size()); pagina++) {
        bool posible = true;
        for (const auto &ingrediente : libro[pagina])
            posible &= esta(ingrediente, heladera_sin_repetidos);
        if (posible)
            realizables.push_back(1+pagina);
```

```
    }
    return int(realizables.size());
}
```

3.2.3. Problema 3: Organizando el Librero [librero2]

http://juez.oia.unsam.edu.ar/#/task/librero2/statement

La solución a este problema se explica en la versión de nivel 3.

Ver librero3 en (3.3.2).

3.2.4. Problema 4: Aprovechando Comodines [comodines]

http://juez.oia.unsam.edu.ar/#/task/comodines/statement

Vamos a resolver incrementalmente las subtareas. Preliminarmente, notemos que la grilla de cartas se puede representar como un grafo G, donde los nodos son las cartas y las aristas conectan *cartas vecinas* (cartas que están arriba, abajo, a la izquierda o a la derecha una de la otra). Diremos finalmente que el *color* de un nodo del grafo es el número escrito en su carta correspondiente.

En este lenguaje de grafos, lo que nos pide el problema es el máximo subgrafo $H \subseteq G$ del grafo original tal que:

1. Sea conexo, es decir que todo par de nodos de H está conectado por un camino de H.
2. Todos los nodos de H que no tengan color 0 tengan el mismo color.

3.2.4.1. Subtarea 1

Para esta subtarea, la condición 2) se simplifica al pedirnos que nuestro subgrafo H tenga nodos todos del mismo color. Esto simplifica nuestro razonamiento de la siguiente manera:

Notemos que ninguna arista de G que conecte nodos de distinto color puede estar en H, por lo que podemos eliminarlas del grafo sin problemas, obteniendo el grafo G', que cumple $H \subseteq G'$. Además, una vez que hacemos esto, toda *componente conexa* de nuestro grafo modificado G' será del mismo color, justamente porque no puede haber caminos entre nodos de distinto color.

Más aún, como le pedimos a H que sea conexo, H no puede ocupar más de una componente conexa, por lo que H debe estar contenido completamente en una

componente conexa. Pero notemos que las componentes conexas de G' son en sí todos valores válidos para H (son conexos y tienen nodos del mismo color). Por lo que el H que estamos buscando no es más que la máxima componente conexa del grafo G'.

Análisis de Complejidad

1. Construir el grafo G se puede hacer en tiempo lineal en la entrada (por cada carta se agrega un nodo, y a lo sumo cuatro aristas a sus vecinos).

2. Eliminar las aristas entre nodos de distinto color se puede hacer también en tiempo lineal al grafo G, recorriendo la lista de adyacencia de cada nodo.

3. Finalmente, encontrar las componentes conexas de G' se puede hacer en tiempo lineal usando, por ejemplo, dfs (http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/dfs).

3.2.4.2. Subtareas 2 y 3

Para estas subtareas se rompe la idea que teníamos para la subtarea 1: ya no es verdad que H sólo contiene aristas que van entre dos nodos del mismo color.

Una manera de arreglar esto es notar que al jugador siempre le conviene hacer que todos los comodines sean del mismo color: el color del que selecciona cartas. Por lo tanto, podemos probar los K colores posibles que el jugador le puede dar a los comodines ($K = 10$ para la subtarea 3 y $K = 1000$ para la subtarea 2), y para cada uno ejecutar el algoritmo de la subtarea 1, cambiando los 0 por el color elegido en cada paso. Al final se tomaría el máximo valor encontrado.

Análisis de Complejidad

El algoritmo de la subtarea 1 es lineal en la entrada, es decir tiene complejidad $\mathcal{O}(N \cdot M)$, y lo ejecutamos K veces, por lo que la complejidad final será $\mathcal{O}(N \cdot M \cdot K)$, que tanto en la subtarea 2 como 3 es suficiente.

3.2.4.3. Subtarea 4

Para la versión final, vamos a tratar de hacer lo que hicimos en el caso anterior pero de manera más inteligente, sin recomputar todo entre distintos colores posibles para los comodines.

Supongamos que construimos el grafo G' y le calculamos las componentes conexas: tenemos algunas componentes que son de colores positivos y algunas otras

que son de color 0. Por cada componente de color positivo me puedo preguntar. ¿Qué pasaría si todos los comodines fueran de mi color? ¿Qué tan grande se volvería mi componente conexa? Para responder esto eficientemente vamos a construirnos un grafo con las componentes conexas de G'.

En este nuevo grafo, que llamaremos Q, los nodos se corresponden a componentes conexas de G', y ponemos una arista entre dos nodos correspondientes a componentes conexas $A, B \subseteq G'$ si había alguna arista que las conectaba en el grafo original G. Diremos además que el *peso* de cada nodo será el tamaño de su componente conexa correspondiente.

Notemos que subgrafos conexos de Q corresponden a subgrafos conexos de G, por lo que en definitiva el grafo Q es muy muy similar a nuestro grafo original G: tiene nodos de distintos colores, algunos positivos y otros 0, y queremos cambiar los 0s por un color positivo para maximizar la máxima componente conexa (en el caso de Q, queremos maximizar el peso total de la componente conexa).

¿De que nos sirvió esta reducción entonces? Pues el grafo Q, por como lo definimos, no puede jamás tener una arista entre dos nodos del mismo color, que es algo que nos va a servir para demostrar la cota de complejidad final.

Volvamos a las preguntas que teníamos antes: *Por cada componente de color positivo me puedo preguntar. ¿Qué pasaría si todos los comodines fueran de mi color? ¿Qué tan grande se volvería mi componente conexa?*

Bueno, pues si me paro en una componente conexa de color positivo y voy recorriendo con dfs los nodos de mi mismo color en el grafo Q, haciendo de cuenta que todos los nodos de color 0 son de mi color, y en el proceso sumo el peso de todos los nodos que descubro, entonces estoy calculando efectivamente el tamaño que hubiese tenido mi componente conexa si todos los comodines tuviesen mi color.

¿Cómo puedo hacer esto en la complejidad adecuada? Si hiciéramos el dfs a lo bruto, y chequeamos en cada paso si el nodo donde estamos parados tiene color adecuado, entonces la complejidad puede volverse cuadrática. Entonces tenemos que hacer algo más inteligente: cuando estamos parados en un nodo sólo debemos mirar los vecinos que tengan color adecuado. Esto lo podemos hacer manteniendo en cada nodo de Q un `map<int, int>` que guarde los vecinos agrupados de a color, en vez de una lista de adyacencia.

Haciendo esto, y teniendo cuidado de no visitar dos veces ningún nodo de Q de color positivo, podemos tomar máximo de todos los tamaños que hallamos y así obtener el resultado al problema.

3.2.4.4. Análisis de Complejidad

Notemos que cuando hacemos el dfs que mencionamos, cada arista se recorre *a lo sumo una vez*, ya que al menos un extremo de la arista tiene color positivo, y la recorrimos necesariamente en el dfs en el que recorrimos ese extremo, y nunca más. Por lo tanto, la complejidad será lineal en la entrada, multiplicado por un factor logarítmico dado por el `map`, con lo que la complejidad final queda: $\mathcal{O}(NM \cdot \log NM)$

3.3. Nivel 3

3.3.1. Problema 1: Jugando Generala [generala]

http://juez.oia.unsam.edu.ar/#/task/generala/statement

3.3.1.1. Solución

Hay muchas maneras de resolver este problema. Una sencilla es programar una función que reciba 6 números, 5 de los dados y uno más, x, y que la función nos diga cuántos dados muestran el número x. Una manera de hacer esta función por ejemplo es haciendo 5 sentencias IF, preguntando si a es igual a x, si b es igual a x, y así con todos, y por cada vez que el programa entra al IF, porque la respuesta fue positiva, sumar uno a una variable **contador** que empieza en 0. Y finalmente devolver el valor de esta variable.
Entonces, con esa función, directamente ya tenemos los primeros 6 resultados, llamando a esa función con $x = 1, 2, ..., 6$. Para saber si hay generala sería preguntar si para alguno de esos valores la función da 5, para el póker es si para alguno de esos valores da 4 o más. Para el full y la escalera es un poco más complejo el asunto.
Pero lo que quiero con esta solución es que conozcan una manera de usar alguna **estructura de datos** un poco más compleja que simplemente INTs. Una estructura de datos es simplemente una manera de guardar información. Por ejemplo, la información de qué tengo en la heladera la puedo guardar como varias palabras simplemente que describan lo que tengo (y estas palabras a su vez podrían estar en orden alfabético o no); o puedo escribir dos columnas, una con el nombre del producto y un número con la cantidad de cuánto tengo; O puedo hacer una tabla donde las columnas representen productos y las filas cantidades, y marcar con una X lo que corresponda; O puedo hacer una tabla para lácteos, una para carnes, etcétera, y agrupar por categoría.
En fin, así como en la vida real hay muchas maneras de guardar información para luego consultarla, en los programas también podemos guardar la información de distintas maneras. El enunciado nos dice que guarda los valores de los dados como

5 variables numéricas, una manera que parece dificultar un poco encontrar las respuestas que queremos.

Entonces, lo que vamos a usar es un arreglo de enteros, que no es nada más que un conjunto de números. Pero ojo, vamos a llevarlo un poco más lejos y en vez de guardar en este conjunto, los valores de los dados, vamos a guardar cuántos tenemos de cada uno, en orden. ¿Por qué? Porque nos va a facilitar las búsquedas que queremos. Podríamos guardar en el arreglo los valores de los dados? Sí, y el código probablemente quedaría un poco más engorroso. Pero esto cada persona lo puede hacer a su manera. Acá simplemente mostramos una manera de hacerlo.

Si el problema pidiera distintas cosas, capaz nos convendría guardar en el conjunto directamente los valores de los dados; pero como vimos al principio, algo que nos va a ser útil es saber la cantidad de dados que tienen un número x.

Pueden leer sobre la estructura **vector** de C++ acá: http://wiki.oia.unsam.edu.ar/curso-cpp/contenedor-vector .

Entonces, supongamos que tenemos un vector, V, que en la posición 1 (no vamos a usar la posición 0 que sería la primera para que no sea confuso) tenemos la cantidad de dados que tienen el 1, en la posición 2 tenemos la cantidad de dados que muestran el número 2, etcétera. Ahora las primeras respuestas se transformaron en simplemente $V[1], V[2] * 2, V[3] * 3, ..., V[6] * 6$.

Para la generala y para el póker, podemos ver si para algún x entre 1 y 6, vale que $V[x] == 5$ ó $V[x] >= 4$ respectivamente. Y esto lo podemos hacer o bien viendo uno por uno, o bien aprovechar que existen las sentencias FOR o WHILE.

Ahora, para la escalera, podemos hacer básicamente dos cosas: Probar cada situación posible, es decir, ver en una condición que $V[1], V[2], V[3], V[4], V[5]$ sean todos 1, en otra condición ver lo mismo para $2, 3, 4, 5, 6$ y lo mismo para $1, 3, 4, 5, 6$. O lo que podemos hacer, es darnos cuenta que para que haya una escalera tienen que ser todos los cinco números distintos, y el número que no esté tiene que ser $1, 2$ ó 6. Entonces podemos, con un FOR, ver que todos los números del vector V, que contabiliza **cuántos dados hay** de cada número, tenga un 1 ó un 0 en todos los valores, y después ver que el 0 esté en alguna de las posiciones $1, 2$ ó 6.

Y para el full, podemos recorrer de vuelta nuestro vector y ver que haya exactamente un valor de nuestro vector que sea 2, y un valor que sea 3. ¿Cómo? Una manera es tener dos variables por ejemplo de tipo *boolean*, una que nos diga si hay algún 2 en V, y la otra que nos diga si hay un 3 en V. Si ambas son **true**, como en total hay 5 dados, sabemos que no puede haber otro dado de más y sí o sí tenemos full.

Dejo un código que hace esto de tener el vector V, pero pueden probar hacer el programa a su manera y después comparar y ver que ¡hay distintas formas de hacer lo mismo!

3.3.1.2. Código

```
#include<iostream>
#include<vector>

using namespace std;

vector<int> generala(int a, int b, int c, int d, int e){
    vector<int> v(7, 0);
    v[a]++; // si a vale 3 por ejemplo, en la posición 3 sumo 1
            // para guardar cuantos hay de cada numero
    v[b]++;
    v[c]++;
    v[d]++;
    v[e]++;

    vector<int> ans;
    ans.push_back(v[1]);
    ans.push_back(v[2]*2);
    ans.push_back(v[3]*3);
    ans.push_back(v[4]*4);
    ans.push_back(v[5]*5);
    ans.push_back(v[6]*6);
    bool deAlgunoHay2=false, deAlgunoHay3=false;
    bool deAlgunoHay4=false, deAlgunoHay5=false;
    bool deNingunoHayMasDe1=true;
    for(int i=1; i<=6; i++){
        if(v[i]>1)deNingunoHayMasDe1=false;
        if(v[i]==2)deAlgunoHay2=true;
        if(v[i]==3)deAlgunoHay3=true;
        if(v[i]==4)deAlgunoHay4=true;
        if(v[i]==5)deAlgunoHay5=true;
    }
    if(deNingunoHayMasDe1 && (v[1]==0 || v[2]==0 || v[6]==0)){
        ans.push_back(20);
    }else{
        ans.push_back(0);
    }
    if(deAlgunoHay2 && deAlgunoHay3){
```

```
        ans.push_back(30);
    }else{
        ans.push_back(0);
    }
    if(deAlgunoHay4 || deAlgunoHay5){
        ans.push_back(40);
    }else{
        ans.push_back(0);
    }
    if(deAlgunoHay5){
        ans.push_back(50);
    }else{
        ans.push_back(0);
    }
    return ans;
}

// ACA PONGO EL EVALUADOR LOCAL
// EN EL JUEZ YA VENIA PROGRAMADO
// LO AGREGO PARA QUE PUEDAN PROBAR TODO EL CODIGO JUNTO SI QUIEREN

int main() {
    int a, b, c, d, e;
    cin>>a>>b>>c>>d>>e;
    vector<int> ans = generala(a, b, c, d, e);
    for(int i=0; i<10; i++){
        cout<<ans[i]<<" ";
    }
}
```

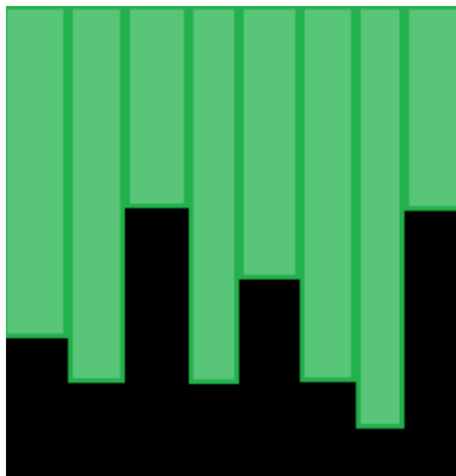
3.3.2. Problema 2: Organizando el Librero [librero3]

http://juez.oia.unsam.edu.ar/#/task/librero3/statement

Este problema se tomó en varios niveles con la misma estructura general de problema y subtareas. La idea general de solución es la misma, pero en los niveles menores se pide computar menos detalles (en nivel 2 no se pide la cantidad total de formas, y en nivel 1 tampoco se pide dar una configuración de ejemplo), con lo cual para esos casos se puede omitir parte de las explicaciones y calcular exclusivamente

lo que se pide.

Si miramos la figura mostrada en el enunciado, lo que podemos observar es que una vez colocados los libros sobre las bases, el hecho de que todos queden alineados a la misma altura significa que los libros junto con las bases terminan formando un **rectángulo**.



Si suponemos que todos los libros tienen el mismo ancho 1, como en el dibujo, este rectángulo que se forma tiene un área total igual a la suma de las alturas de los libros (área verde) más la suma de las alturas de las bases (área negra). Además, la base del rectángulo es N, la cantidad de libros de y de bases. Si llamamos h a la altura del rectángulo, sabiendo que el área del rectángulo se obtiene como base por altura, tendremos entonces:

$$N \cdot h = s_{libros} + s_{bases}$$

Donde s_{libros} es la suma de las alturas de los libros, y s_{bases} la suma de las alturas de las bases.

De aquí podemos despejar h a partir de datos que vienen en la entrada:

$$h = \frac{s_{libros}+s_{bases}}{N}$$

Además se observa de esto que si $s_{libros} + s_{bases}$ no resulta múltiplo de N, será imposible lograr la tarea y podemos retornar altura -1 e indicar que hay 0 formas.

Uno puede pensar que esto ya calcula la altura correcta y por lo tanto resuelve el problema nivel 1. Sin embargo, solo resuelve algunas subtareas, ya que esa será la altura resultante, **suponiendo que sea posible** formar este rectángulo. Puede ser que no exista manera de emparejar los libros con las bases para obtener un librero alineado.

Lo que sí podemos observar una vez que tenemos despejado h, es que

necesariamente cada libro con altura x tiene que ir sobre una base de altura $h - x$, para que la altura total sea siempre h. Como $h - x$ decrece a medida que x crece, esto implica que necesariamente, si es posible obtener un librero bien alineado, puede obtenerse ordenando los libros ascendentemente y las bases descendentemente.

Esto facilita mucho la implementación: ordenamos ambos arreglos de entrada, con los criterios indicados, y luego recorremos verificando que para todo i, la altura de la base i-ésima y la del libro i-ésimo sumen h. Si esto ocurre, la altura h funciona, y sino, es imposible y se debe retornar -1. Ahora sí, tenemos una solución de 100 puntos para el problema nivel 1.

Para el problema nivel 2, la idea es exactamente la misma, pero se suma la complejidad adicional de retornar un arreglo que indique cómo ubicar los libros en el librero. Igualmente, si implementamos esta solución nivel 1 que simplemente calcula la altura, obtenemos 60 puntos.

Una forma de implementar este cómputo del ordenamiento, es ampliar nuestra noción de libro y base: En lugar de guardar en los arreglos que ordenamos únicamente las alturas, podemos tener pares con la altura y el índice original de la base o libro correspondiente. De esta forma, al ordenar por altura, igualmente mantenemos el índice original en donde se ubicaba cada libro o base antes de aplicar el ordenamiento.

Entonces para construir el arreglo final *orden*, podemos recorrer nuestros arreglos ordenados y, al ver el i-ésimo libro y la i-ésima base (en orden de altura), si estos tenían respectivamente los índices originales i_{libro} e i_{base} en la entrada (contando desde cero), hacemos $orden[i_{base}] \leftarrow 1 + i_{libro}$, para indicar que en esa base se ubica ese libro. Corresponde poner el +1 simplemente porque se nos pide numerar los libros desde 1, pero en la implementación es más cómodo tenerlos desde 0.

En nivel 3, todo esto permite obtener 60 puntos, pero para tener 100 puntos se debe además calcular la cantidad de formas posibles de organizar el librero. Ya hemos visto cómo saber si es posible o no: en los casos en que no es posible, retornaremos altura -1 y cantidad de posibilidades 0.

Suponiendo que es posible, hay que pensar cómo son las distintas soluciones que existen. En cada una de ellas, lo único importante es que a los libros de altura x los ubiquemos sobre bases de altura $h - x$. Por lo tanto, para que sea posible va a ocurrir necesariamente que para cualquier x, hay la misma cantidad de libros de altura x, que bases de altura $h - x$. Y si esa cantidad es t, la cantidad de formas posibles de reordenar esos t libros en esas t bases es $t \cdot (t-1) \cdots 2 \cdot 1 = t!$[5].

[5] https://es.wikipedia.org/wiki/Factorial

Por lo tanto, la cantidad total de formas se obtiene multiplicando, para cada valor posible de x (las **distintas** alturas de los libros), el correspondiente valor $t!$. La multiplicación de todos estos factoriales da por resultado la respuesta total. Hay que tener cuidado de hacer todas las multiplicaciones módulo 1.000.000.007, para evitar tener overflow. Esto permite finalmente obtener 100 puntos en nivel 3.

3.3.3. Problema 3: Ubicando fichas en la hilera [hilera]

http://juez.oia.unsam.edu.ar/#/task/hilera/statement

3.3.3.1. Subtarea 1

En este caso, $K = 0$ y $N \leq 2$. Es decir, no tenemos fichas especiales y hay a lo sumo dos números en la hilera. Si $N = 1$, podemos cubrir al único número de la hilera con una ficha, y esa será la solución. Si $N = 2$, hay que cubrir al más grande, ya que no es posible cubrir a ambos. Se puede escribir la solución que obtiene 4 puntos utilizando ifs directamente:

```
long long hilera(vector<int> numeros, int k, vector<int> &fichas) {
    fichas.clear();
    if (N == 1) {
        fichas.push_back(1);
        return numeros[0];
    } else { // N == 2
        if (numeros[0] > numeros[1]) {
            fichas.push_back(1);
            fichas.push_back(0);
            return numeros[0];
        } else {
            fichas.push_back(0);
            fichas.push_back(1);
            return numeros[1];
        }
    }
}
```

3.3.3.2. Subtarea 2

En cada una de las N posiciones, tenemos tres opciones (codificadas como 0, 1 o 2 en la salida): No poner ficha, poner una ficha común, o poner una ficha especial. Eso da un total de 3^N posibilidades, y como $N \leq 9$ en esta subtarea, es posible

explorar exhaustivamente todas ellas utilizando directamente la técnica de fuerza bruta (http://wiki.oia.unsam.edu.ar/algoritmos-oia/backtracking).

En este caso, podríamos poner 9 fors que iteren los valores 0, 1 , 2 para probar todas las posibilidades. Cuando N sea menor que 9, muchos valores no se usarán, pero en el código es más fácil poner siempre 9 de estos fors, para estar cubiertos ante el caso máximo. También es posible programar esto de otras maneras más avanzadas y elegantes, pero mostraremos solamente la forma directa con for.

Una vez que tenemos ya seleccionada una configuración de valores 0,1,2 dentro de los fors, hay que verificar si es válida: no tiene que haber dos 1 pegados, y puede haber como máximo K valores 2. En ese caso, sumamos los valores de los números que no tengan 0, y ese es el valor obtenido. De todas las soluciones, guardamos en una variable el máximo conseguido.

El código quedaría:

```
#define qforn(i,n) for(i = 0; i < int(n); i++)
#define forn(i,n) for(int i = 0; i < int(n); i++)

long long hilera(vector<int> numeros, int k, vector<int> &fichasSolucion) {
    long long mejorValor = -1;
    // Siempre ponemos 9 lugares y 9 fors, mas facil
    vector<int> fichas(9);
    qforn(fichas[0], 3)
    qforn(fichas[1], 3)
    qforn(fichas[2], 3)
    qforn(fichas[3], 3)
    qforn(fichas[4], 3)
    qforn(fichas[5], 3)
    qforn(fichas[6], 3)
    qforn(fichas[7], 3)
    qforn(fichas[8], 3) {
        bool sinUnosPegados = true;
        forn(i,8)
        if (fichas[i] == 1 && fichas[i+1] == 1)
            sinUnosPegados = false;
        int total2 = 0;
        forn(i,9)
        if (fichas[i] == 2)
            total2++;
        if (sinUnosPegados && total2 <= K) {
            long long total = 0;
            forn(i,numeros.size())
            if (fichas[i] > 0)
                total += numeros[i];
            if (total > mejorValor) {
                mejorValor = total;
                fichasSolucion = fichas;
            }
        }
    }
    fichasSolucion.resize(numeros.size()); // Borramos el sobrante
    return mejorValor;
}
```

Con esta solucion se obtienen 16 puntos en total.

3.3.3.3. Subtarea 3

En este caso se garantiza que $K = N$. Es una subtarea en la cual tenemos a nuestra disposición muchas fichas especiales: de hecho, como tenemos N, podemos poner una por cada posición de la hilera. Esto nos garantiza poder sumar todos los números, que es lo mejor posible, así que esa es una solución válida.

Basta entonces que el programa retorne la suma de los números, y cargue en el arreglo de fichas N valores 2, para resolver esta subtarea y obtener 4 puntos. Si lo combinamos con las anteriores, ya obtendríamos 20 puntos en total.

3.3.3.4. Subtarea 4

En esta subtarea se nos garantiza $N = 2K + 2$. Notar que eso significa que N será par. Y esto también equivale a que $K = \frac{N}{2} - 1$.

La clave de esta subtarea es que con esa cantidad de fichas especiales, podemos cubrir todos los números menos uno cualquiera que elijamos. Esto porque con $\frac{N}{2}$ fichas comunes puestas en las posiciones pares (o en las impares) cubrimos uno de cada dos números, sin poner dos adyacentes. Y luego podemos cubrir libremente otros $K = \frac{N}{2} - 1$ números usando las fichas especiales, con lo cual en total cubrimos $\frac{N}{2} + (\frac{N}{2} - 1) = N - 1$ posiciones.

Siempre conviene excluir al número más pequeño, pues eso hará la suma de los incluidos lo más grande posible. La respuesta en este caso es la suma de todos los números, menos el más pequeño de ellos. Escribir un código para este caso nos agrega 16 puntos más.

3.3.3.5. Subtarea 5

En esta subtarea, N es grande (hasta 5.000) pero $K = 0$: es decir, no hay fichas especiales, o sea que únicamente podemos poner fichas comunes, y en la solución de salida únicamente aparecerán los números 0 y 1.

Como todos los números que elijamos tendrán encima una ficha común, podremos elegir cualquier subconjunto de números de la hilera, siempre y cuando respetemos la condición de no elegir dos adyacentes.

Este problema puede resolverse muy eficientemente con programación dinámica. Recomendamos conocer lo básico de esta técnica antes de leer esta solución (`http://wiki.oia.unsam.edu.ar/algoritmos-oia/programacion-dinamica`).

Para esto, vamos a llamar $f(i)$, para $0 \leq i \leq N$, a la máxima suma que podemos conseguir si ponemos las fichas de forma ideal **limitados a los primeros** i **números de la hilera**.

La respuesta al problema será justamente $f(N)$, así que si podemos calcular estos valores f (de los cuales hay $N + 1$ valores para calcular: el $f(0)$, el $f(1)$, el $f(2)$, y así siguiendo hasta el $f(N)$), tendremos la máxima suma posible.

Veamos primero que los valores pequeños son simples: $f(0) = 0$, ya que si miramos los primeros 0 números, es imposible sumar nada, así que tenemos suma 0 en ese caso.

A su vez, si llamamos v al arreglo de números, de forma que los números de la hilera en orden son $v_0, v_1, \cdots, v_N$, podemos ver que $f(1) = v_0$: ya que si nos concentramos en resolver el problema solamente para el primer número, no hay con quien pueda chocar, así que lo mejor es elegirlo y listo, esa será la suma máxima (Pero notar que ¡esto solamente es cierto gracias que los números nunca son negativos!).

Ahora hay que resolver la situación y calcular $f(i)$ para $i \geq 2$. La clave está en dividir en los únicos dos casos posibles:

1. Ponemos una ficha sobre el número v_{i-1} (el último de los primeros i números que estamos considerando). En este caso, ya tenemos garantizado al menos v_{i-1} para nuestra suma final. Haber puesto la ficha allí hace que ahora **esté prohibido** ubicar una ficha sobre v_{i-2}. Por lo tanto, todas las demás fichas se ubicarán sobre los primeros $i - 2$ números de la hilera (los anteriores a v_{i-2}: desde v_0 hasta v_{i-3} inclusive). ¡Y ninguna de esas posiciones es adyacente a la ficha que hemos puesto sobre v_{i-1}! Eso hace que cualquier forma válida de ubicar fichas mirando solamente esas primeras $i-2$ posiciones, sea compatible o extensible a todas las primeras i posiciones: simplemente le agregamos la ficha que ponemos sobre v_{i-1}. La suma total será la suma que tengamos sobre las primeras $i - 2$ posiciones, más v_{i-1}. Por lo tanto siempre nos conviene tomar la mejor suma posible usando las primeras $i - 2$ posiciones. Y esto es justamente el valor $f(i - 2)$.

2. **NO** ponemos una ficha sobre el número v_{i-1}. En este caso, como no vamos a poner una ficha sobre ese número, es lo mismo exactamente que si no existiera. Por lo tanto lo mejor que podemos hacer es lo mismo que si solo mirásemos los primeros $i - 1$ números. Y eso ya lo tenemos calculado en $f(i - 1)$.

Del análisis anterior, tenemos que en el caso 1, tendremos el valor $f(i-2)+v_{i-1}$. Y en el caso 2 tendremos el valor $f(i - 1)$. ¿Cómo sabemos si la solución óptima

pone una ficha sobre el número v_{i-1} o no? Simplemente comparamos estos valores y tomamos el mayor, ya que así obtendremos el máximo valor posible en cualquier caso, que es justamente el valor óptimo que queremos.

Nos queda entonces la recursión:

$f(i) = \text{máx}(f(i-2) + v_{i-1}, f(i-1))$

Que implementada en forma directa como función recursiva, tomará un tiempo exponencial en ejecutar. Pero si en lugar de eso declaramos un vector de $N+1$ posiciones, digamos p, y con un for lo iteramos cargando en $p[i]$ el valor correspondiente según la fórmula anterior, podemos calcular todos los valores en solamente $O(N)$ pasos. Esto en código sería algo así:

```
vector<long long> p(N+1);
p[0] = 0;
p[1] = v[0];
for (int i=2; i<=N; i++)
    p[i] = max(p[i-2] + v[i-1], p[i-1]);
```

Notar que **por el orden en que recorremos**, siempre que vamos a llenar una posición i con su valor, están ya calculadas anteriormente aquellas que son necesarias para el cálculo (en este caso, son las posiciones $i-1$ e $i-2$, que ya estarán calculadas al llegar a i porque vamos de izquierda a derecha). Notar también que hemos tenido que agregar casos base para 0 y 1, ya que así evitamos salirnos de rango al acceder a las posiciones $i-1$ e $i-2$.

Esta solución de programación dinámica por sí sola permite obtener 28 puntos (resuelve las subtareas con $K = 0$, que son la 1 y la 5), y combinada con las ideas anteriores podemos llegar a 60 puntos.

3.3.3.6. Subtarea 6

La subtarea 6 es la más difícil. Puede resolverse con un algoritmo de programación dinámica un poco más complicado que el anterior, y que al mismo tiempo resuelve todas las subtareas, con lo cual si un participante programa correctamente esta solución completa de 100 puntos, ya no necesita programar ninguna otra.

La clave es definir una función f similar a la de la subtarea anterior, pero que tenga en cuenta la cantidad K de fichas especiales disponibles. Podemos definir[6]

[6]Esta no es la única función posible: en casi todos los problemas de programación dinámica,

$f(i, K)$, como la máxima suma que es posible conseguir ubicando fichas sobre los i primeros números[7], **pero teniendo en cuenta** que entre las fichas que ubicamos podemos usar a lo sumo K fichas especiales.

Esto cambia la fórmula recursiva porque ahora hay 3 casos para calcular el valor de $f(i, k)$:

1. Si no ubicamos ficha sobre v_{i-1}, tenemos en forma casi igual que antes una suma máxima $f(i-1, k)$. El valor k ahora lo mantenemos intacto, ya que no cambia en absoluto la situación de que podemos poner k fichas especiales como máximo, si decidimos no poner ninguna en absoluto sobre el último número considerado.

2. Si ubicamos una ficha común sobre v_{i-1}, estaríamos tentados de copiar nuevamente la fórmula como antes: $v_{i-1} + f(i-2, k)$, y el k se mantiene porque no hay ningún cambio a la situación de las fichas especiales. **Pero esto tiene un error:** ¿Qué pasaría si lo mejor fuese poner una ficha común sobre v_{i-1}, y una especial sobre v_{i-2}? En esta fórmula, al pasar a mirar los primeros $i-2$ números directamente, estamos suponiendo que no vamos a poner **ninguna ficha** sobre el anteúltimo número, así que esta solución no estaría considerada, y el programa fallaría cuando esta opción sea la mejor (ejercicio: ¡escribir un ejemplo donde esta opción sea la única forma óptima!). Lo que ocurre es que en la subtarea anterior, al no haber especiales, es cierto que en cualquier solución válida en que ponemos una ficha en el último número, no podemos poner ninguna sobre el anteúltimo. Pero ahora que hay fichas especiales, se nos abren dos posibilidades distintas:

   a) Efectivamente no poner ninguna ficha sobre v_{i-2}. Y en este caso, sí tenemos como antes, $v_{i-1} + f(i-2, k)$

   b) Poner una ficha sobre v_{i-2}. Esto solamente será posible con una ficha especial (y por lo tanto, solamente hay que considerar esta posibilidad cuando $k > 0$), porque si usamos una común, tendremos dos fichas comunes adyacentes (en posiciones $i-1$ e $i-2$). Luego de poner una ficha común en $i-1$ y una especial en $i-2$, no queda ninguna restricción para las primeras $i-2$ posiciones: cualquier ubicación válida allí, se puede extender con las fichas especial y común que hemos propuesto, y sigue siendo válida. La respuesta entonces será en este caso $v_{i-1} + v_{i-2} + f(i-2, k-1)$. Notar que usamos $k-1$ porque entre esas primeras $i-2$ posiciones, solamente

existen muchas variantes que se pueden hacer de la función, tales que aún así todas ellas resuelvan el problema correcta y eficientemente.

[7] Hasta aquí, es exactamente igual que en la subtarea anterior

podemos permitir $k-1$ fichas especiales, ya que vamos a colocar una en $i-2$ y entonces lo máximo que podemos poner en las primeras $i-2$ sin pasarnos en total de la cantidad k permitida, es $k-1$.

3. Finalmente, podríamos ahora poner directamente una ficha especial sobre v_{i-1} (si es que $k>0$). En este caso, queda $v_{i-1}+f(i-1,k-1)$

Como siempre, no sabemos cuál de estos 4 casos será el mejor, así que tomamos el máximo de todos ellos. Algunos casos solo se consideran si $k>0$.

A su vez, como la f ahora tiene dos parámetros, hay que guardar sus valores en una matriz, en lugar de en un arreglo. Por ejemplo si la guardamos en la matriz p, podríamos guardar el valor $f(i,k)$ en $p[i][k]$, e ir calculando esta matriz de a filas, ya que por la fórmula en cada paso solamente se necesitan las dos filas anteriores para hacer los cálculos.

Esta solución tiene complejidad $O(NK)$, y es suficiente para obtener 100 puntos en este problema.

3.3.4. Problema 4: Ubicando la oficina de correo [correocentral]

http://juez.oia.unsam.edu.ar/#/task/correocentral/statement

3.3.4.1. Prerequisitos

- Grafos Dirigidos
- DFS
- vector de C++

3.3.4.2. Modelado

Si ya leímos suficientes problemas de este tipo, nos daremos cuenta que las N ciudades y M vías mensajeras constituyen un grafo dirigido.

En este problema nos hablan de seguidillas de vías mensajeras, denominadas rutas mensajeras. Esto no es más que un camino en el grafo. Además si existe una ruta (o camino) entre dos ciudades A y B, y también existe un camino entre B y A, decimos que A y B están comunicadas o, en términos de grafos, son dos nodos fuertemente conectados. Además una ciudad está comunicada con si misma.

Queremos saber cuál es la máxima cantidad de ciudades con las que una ciudad se comunica. O en términos de grafos: ¿Cuál es la máxima cantidad de nodos con las que un nodo se conecta fuertemente?

Vamos a introducir el concepto de *componente fuertemente conexa*[8]. Una componente es un conjunto de nodos. Dos nodos pertenecen a la misma componente fuertemente conexa si y solo si están fuertemente conectados. Es decir que dos nodos pertenecen a la misma componente si y solo si hay forma de ir y volver entre los dos nodos.

Ahora un poco de teoría. Cualquier grafo se puede particionar en componentes fuertemente conexas. Es decir una partición donde cualquier nodo pertenece a exactamente una componente fuertemente conexa. Además esta partición es única.

Volviendo a nuestro problema, queríamos saber cuál es la máxima cantidad de nodos con las que un nodo se conecta fuertemente (incluyendo a si mismo). Es decir, queremos ver cuál es el tamaño de la componente fuertemente conexa más grande de un grafo.

Para resolver el problema vamos a utilizar un algoritmo que encuentre la partición de nodos en componentes fuertemente conexas. Luego simplemente buscamos cuál es la componente más grande.

3.3.4.3. Soluciones

Una primera solución bastante fácil de programar tiene como idea realizar múltiples DFS. Esta solución tiene una complejidad temporal más alta que la solución completa, es decir que nos dará un puntaje parcial.

En esta solución, lanzamos un DFS desde cada nodo (u) del grafo que permite saber que nodos son alcanzables desde el nodo u. Almacenamos esta información en una matriz `alcanzable`.

La matriz `alcanzable` es una matriz de booleanos, donde la casilla `alcanzable[u][v]` es verdadera si y solo si existe un camino de u a v.

Luego determinamos que u y v pertenecen a la misma componente fuertemente conexa si y solo si son verdaderas las casillas `alcanzable[u][v]` y `alcanzable[v][u]`.

Complejidad: $\mathcal{O}\left(\#\text{nodos} \cdot (\text{DFS de todo el grafo})\right) = \mathcal{O}\left(n \cdot (n + m)\right)$

[8] http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos-dirigidos/componentes-fuertemente-conexas-en-dirigidos

```
typedef struct {
  const int n;
  // Lista de adyacencia
  const vector<vector<int> >& ady;
} grafo;

void marcar_alcanzables(const graph& g, int from,
                        int u,
                        vector<vector<bool> >& alcanzable) {
  if (alcanzable[from][u]) {
    return;
  }

  alcanzable[from][u] = true;

  for (int v : g.ady[u]) {
    marcar_alcanzables(g, from, v, alcanzable);
  }
}

int max_scc(const graph& g) {
  vector<vector<bool> > alcanzable(
        g.n + 1,
        vector<bool> (g.n + 1, false));

  forn(u, g.n+1) {
    marcar_alcanzables(g, u, u, alcanzable);
  }

  int max_scc_sz = 0;

  forn(u, g.n+1) {
    /* sz va a ser el tamanio de la scc a la que
       pertenece el nodo u */
    int sz = 0;
    forn(v, g.n+1) {
      if (alcanzable[u][v] && alcanzable[v][u]) {
        sz++;
      }
```

```
    }

    max_scc_sz = max(max_scc_sz, sz);
  }

  return max_scc_sz;
}

int correocentral(int n, vector<int> a, vector<int> b) {
  // El nodo 0 es aislado, no lo tomamos en cuenta;
  int m = int(a.size());

  forn(i, m) {
    ady[a[i]].push_back(b[i]);
  }

  graph g = {n, ady};
  return max_scc(g);
}
```

Para escribir una solución lineal tenemos que utilizar uno de los algoritmos para particionar grafos en componentes fuertemente conexas en complejidad lineal. Hay dos conocidos, el algoritmo de Tarjan y el *algoritmo de Kosaraju*[9]. En este caso vamos a utilizar el último.

En la wiki podemos encontrar una explicación detallada del algoritmo de Kosaraju. Como versión corta, podemos adelantar que el algoritmo da una partición utilizando dos DFS: uno donde establece un orden de recorrido, y el segundo, que utiliza este orden y recorre los ejes invertidos para encontrar las componentes fuertemente conexas.

```
typedef struct {
  // nodos del grafo
  const int n;
  /* Lista de adyacencia
    out_ady[u] lista los nodos que salen de u
```

[9] http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos-dirigidos/componentes-fuertemente-conexas-en-dirigidos

```
  */
  const vector<vector<int> >& out_ady;
  /* Lista de adyacencia en sentido inverso
     in_ady[u] lista los nodos que entran a u
  */
  const vector<vector<int> >& in_ady;
} grafo;

void visitar_y_dar_orden(vector<bool>& visitado,
                         vector<int>& orden,
                         const grafo& g, int u) {
  if (visitado[u]) {
    return;
  }

  visitado[u] = true;
  for (int v : g.out_ady[u]) {
    visitar_y_dar_orden(visitado, orden, g, v);
  }

  orden.push_back(u);
}


void asignar(vector<bool>& fue_asignado,
             const grafo& g,  int u, int& sz) {
  if (fue_asignado[u]) {
    return;
  }
  sz++;
  fue_asignado[u] = true;

  for (int v: g.in_ady[u]) {
    asignar(fue_asignado, g, v, sz);
  }
}


int get_max_scc_sz(const grafo& g) {
```

```
  vector<int> order;
  vector<bool> visitadoRecorrido1(g.n+1, false);
  vector<bool> visitadoRecorrido2(g.n+1, false);

  forn(i, g.n+1) {
    visitar_y_dar_orden(visitadoRecorrido1, order, g, i);
  }

  reverse(order.begin(), order.end());

  int max_scc_sz = 0;
  for (int v: order) {
    int sz = 0;
    // Cuando corremos asignar, se visita toda la scc
    // a la que pertenece v.
    // sz (size, tamanio) lleva la cuenta de los nodos en la scc

    asignar(visitadoRecorrido2, g, v, sz);
    max_scc_sz = max(max_scc_sz, sz);
  }
  return max_scc_sz;
}

int correocentral(int n, vector<int> a, vector<int> b) {
  // El nodo 0 no lo consideramos
  vector<vector<int> > out_ady(n+1);
  vector<vector<int> > in_ady(n+1);
  int m = int(a.size());

  forn(i, m) {
    out_ady[a[i]].push_back(b[i]);
    in_ady[b[i]].push_back(a[i]);
  }
  grafo g = {n, out_ady, in_ady};
  int max_scc_sz = get_max_scc_sz(g);

  return max_scc_sz;
}
```

Capítulo 4

Certamen Nacional

4.1. Nivel 1

4.1.1. Problema 1: Aprendiendo operaciones [aprendiendo]

http://juez.oia.unsam.edu.ar/#/task/aprendiendo/statement

4.1.1.1. Subtarea D = 1

Si tenemos que repartir una cantidad N de caramelos entre 1 persona, cuántas sobran? 0, siempre! Le doy a esa persona todos los caramelos y listo, terminé de repartir.

4.1.1.2. Una idea

Si hay que repartir entre dos personas, pensemos: Nos pueden sobrar 3 caramelos? Si estamos repartiendo uno para cada persona alternadamente, cuando tenemos 3 caramelos, en vez de que nos sobren, le damos uno más a cada una y listo, pudimos seguir repartiendo. Generalizando, nunca nos pueden sobrar una cantidad de caramelos mayor o igual a D. Si estamos repartiendo alternadamente, y en algún momento tenemos más de D, todavía le podemos dar uno más a cada persona, y entonces 'lo que sobra' es menos.

4.1.1.3. Subtareas D = 2, 5, 10, 4, 8, 3, 9: Criterios específicos

Decimos que un número A es divisible por otro número B cuando puedo repartir A caramelos entre B personas y darle la misma cantidad a cada una sin que me sobre ninguno.
Algo que nos va a ayudar con estas subtareas es algo que se conoce como 'criterios de divisibilidad'. Un criterio de divisibilidad es una regla que nos ayuda a saber

cuándo un número es divisible por otro. De hecho, no sólo eso, sino que nos dice, de no ser divisible, cuántos me sobran. No indica cuántos caramelos llego a darle a cada persona, pero sí cuántos sobran.
La razón para estas subtareas es porque los criterios de divisibilidad por estos valores de D son los más conocidos y sencillos de poner en práctica.
Por ejemplo, para saber si un número es divisible por 2, basta con mirar la última cifra: Si es par el número será divisible, y si no el resto será 1. El resto en la división por 10 será igual a la última cifra.
Entonces teniendo a N, un número tan grande que hay que guardarlo como string, pero siendo $D = 10$ por ejemplo, simplemente devolvemos como resto la última cifra y listo.
Los otros criterios se pueden buscar y encontrar, pero ahora voy a contar un poco lo que hay detrás de estas reglas.

4.1.1.4. Un concepto súper importante

Primero veamos algo que se puede explicar sencillamente pero envuelve un concepto súper importante en una gran parte de la matemática.
El resto de dividir $A \cdot B$ por D, es igual al resto de dividir a A por D multiplicado por el resto de dividir a B por D. Y si es producto fuera mayor D, es el resto de ese producto dividido D.

Por ejemplo, el resto de $9 \cdot 8 = 72$ divido 5 es 2, que es igual al resto de $4 \cdot 3 = 12$ dividido 5. (4 y 5 son los restos al dividir 9 y 8 por 5 respectivamente.)
La manera de ver esto es la siguiente: Si $A = m \cdot D + r_1$ y $B = n \cdot D + r_2$, con $0 <= r_1, r_2 < D$ los restos, que son los números que nos importan, y m y n enteros, que es la cantidad de caramelos que le daría a cada una de las D personas. Entonces, $A \cdot B = (m \cdot D + r_1) \cdot (n \cdot D + r_2) = m \cdot D \cdot n \cdot D + m \cdot D \cdot r_2 + n \cdot D \cdot r_1 + r_1 \cdot r_2$. Lo que puedo hacer es agarrar los primeros $m \cdot D \cdot n \cdot D$ caramelos y darle $m \cdot D \cdot n$ a cada persona. Después agarro los $m \cdot D \cdot r_2$ caramelos y le doy $m \cdot r_2$ a cada una (si r_2 es 0 entonces le doy 0 caramelos a cada una y listo, sigo como si nada). Después agarro los $n \cdot r_1 \cdot D$ caramelos siguientes y reparto $n \cdot r_1$ a cada persona. Y por último, me quedan $r_1 \cdot r_2$ caramelos. Si tengo menos que D ya está, ese es mi resto. Y si no, los empiezo a repartir hasta tener menos que D, y por eso lo que me sobre al final será lo que me sobraría si empezara directamente con $r_1 \cdot r_2$.
Se puede verificar que con la suma pasa lo mismo; es decir, el resto de una suma será la suma de los restos (o el resto de la suma de los restos).

4.1.1.5. Una solución usando esto

Ahora que vimos esto, vamos a la segunda y última idea: Pensar al número N cifra a cifra. Así, el 1942 por ejemplo será $1000+900+40+2 = 1 \cdot 1000+9 \cdot 100+4 \cdot 10+2$. Como a N lo tenemos como string debido a lo grande que puede ser el número, no podemos descomponerlo exactamente en cifras y sumar porque la cifra de la izquierda puede venir acompañada de 999 ceros, entonces no nos sirve. Pero lo que podemos hacer es multiplicar de a 10 hasta llegar a la potencia deseada, y cada vez que multiplicamos quedarnos con el resto del producto, porque ya vimos que nos va a quedar el mismo resto si hacemos esto.

Entonces, vamos de derecha a izquierda, y según dónde estemos, agarramos a la cifra correspondiente y la multiplicamos por 10 tantas veces como dígitos hayan quedado a la derecha, tomando el resto de lo que obtenemos cada vez. Y vamos sumando todos estos restos (y podemos quedarnos con el resto también al sumar entre cifras, por lo que vimos antes).

4.1.1.6. Complejidad de esa solución y algo un poco mejor

Veamos que para la cifra de más a la derecha no hacemos mucho. Para la segunda cifra multiplicamos por 10 una vez; para la siguiente cifra 2 veces, luego 3, y así hasta la última cifra, que si N tiene 100 dígitos, habrá que multiplicar por diez 999 veces. Por lo que la cantidad de veces que multiplicamos por 10 es $0 + 1 + 2 + ... + 999 = \frac{999 \cdot 1000}{2}$, que es alrededor de 1000^2. Es decir, si L es la cantidad de dígitos de N, nuestro algoritmo tiene complejidad $O(L^2)$. (Pueden leer más sobre complejidad y esta notación acá: http://wiki.oia.unsam.edu.ar/algoritmos-oia/analisis-amortizado)

Lo que conviene acá, que se usa mucho en problemas de este estilo con números muy grandes dados como strings, es ir de izquierda a derecha. La idea, que para mí es menos intuitiva que simplemente multiplicar por 10 muchas veces, es formar al número desde la izquierda. Al 1942 lo vamos a formar teniendo primero el 1, luego el 19, después el 194 y finalmente el 1942. Y con esto en mente, no es muy difícil entender cómo: En cada paso lo que hacemos es agregar un dígito. Entonces lo que tenemos que hacer es simplemente multiplicar por 10, para tener un 0 a la derecha, y después sumar la cifra que queremos poner al final.

Es un cambio pequeño el de ir de izquierda a derecha pero fíjense que por cada cifra ahora estamos multiplicando por 10 una sola vez! Entonces la complejidad de este algoritmo pasa a ser $O(L)$ en vez de $O(L^2)$.
De la misma manera que antes cada vez que multipliquemos por 10 y sumemos

podemos quedarnos con el resto del resultado y será lo mismo que aplicarlo al final.

4.1.1.7. Extra - para pensar

Los criterios de divisibilidad más conocidos surgen en general de descomponer a los números como sumas de dígitos por potencias de 10. Por ejemplo, el criterio del 3 dice que el resto de un número al ser dividido entre 3 es el resto de la suma de los dígitos. Y eso sale de que el resto de 10 dividido 3 es 1. Entonces, como el resto del producto es el producto de los restos, multiplicar por 10 es como multiplicar por 1, y entonces el resto de $1 \cdot 10^3 + 9 \cdot 10^2 + 4 \cdot 10^1 + 2$ es el resto de $1 \cdot 1^3 + 9 \cdot 1^2 + 4 \cdot 1^1 + 2 = 1 + 9 + 4 + 2$, y lo mismo para cualquier número, las potencias de 10 se reemplazan por unos y listo. Lo mismo pasa con la divisibilidad por 9.

Pueden leer más sobre esto en internet, y encontrar mucho buscando sobre "congruencias" de números, ya que se dice que dos números son **congruentes módulo M** si esos dos números tienen el mismo resto en la división por M.

4.1.2. Problema 2: Computando patentes [patentes]

http://juez.oia.unsam.edu.ar/#/task/patentes/statement

Hay varias maneras de encarar este problema. Una, que es un poco ineficiente pero sencilla de programar, es simplemente hacer varios FORs, uno adentro de otro, y que el FOR recorra o bien las letras o bien los dígitos según corresponda, y formar todas las patentes que existen. Cuando pasamos por la patente que tenemos dada, empezamos a contar, y cuando llegamos a pasar K patentes más, devolvemos la que se forma con las variables que tenemos de los FORs.
Si pensamos en cuántas operaciones haríamos en total, como los FORs están uno adentro de otro hay que multiplicar, es decir por cada vez que movemos una variable de afuera, movemos una cierta cantidad de una de adentro, y por cada una de esas movemos otra cantidad de veces otra de más adentro, y así sucesivamente.
Entonces, para las patentes de 3 letras y 3 dígitos, tenemos $26 \cdot 26 \cdot 26 \cdot 10 \cdot 10 \cdot 10$ operaciones, ya que para cada patente no hacemos nada muy complejo, sólo vemos si la patente que se forma es la que nos dan, y quizás sumamos uno a una variable, la comparamos con K y devolvemos la concatenación de las variables. Por lo que nuestro algoritmo realiza unas $26^3 \cdot 1000$ operaciones, que en términos de potencias de 10 es alrededor de $2 \cdot 10^7$. Para el otro tipo de patentes serán $26^4 \cdot 1000$ que en términos de potencias de 10 es alrededor de $4 \cdot 10^8$ operaciones. Esa complejidad puede ser suficiente en algunos contextos o en algunas competencias, o demasiado tiempo en otras.

De hecho, esta solución es más que suficiente para las subtareas donde K va hasta 100. Tenemos que tener el cuidado de los FORs no empezarlos a todos en A ó 0, sino por ejemplo el primer FOR lo empezamos en la primera letra de la patente; el segundo FOR, lo empezamos dependiendo del anterior FOR, o bien en la segunda letra de la patente dada, si la variable anterior es igual a la primera o bien en A si no. Este razonamiento no es tan sencillo de ver, quiero que se queden acá y lo entiendan con este ejemplo: Si queremos saber la patente que va K números después de, por ejemplo, $CDP123$, la primera letra no tiene sentido que sea ni A ni B. Y, si en el primer FOR, estamos en la opción de que la primera letra sea C, la segunda no va a ser ni A, ni B, ni C, porque estaríamos en patentes anteriores a la dada, y buscamos posteriores.
Teniendo estos cuidados, no vamos a hacer más que K operaciones, ya que arrancamos en la patente y por cómo son los FORs, vamos avanzando de a una patente.
Ahora voy a describir una solución mucho más eficiente.
Supongamos primero que tenemos toda la lista de las patentes, ordenadas. Si nos dan una patente, y K, basta con saber cuál es la posición p donde está la patente en cuestión, y devolver la patente de posición $p + K$.
Pensando en esto, si pudiéramos, dada una patente, saber en qué posición está, y dada una posición, saber qué patente pertenece a ella, tendríamos el problema resuelto, sin necesidad de tener toda la lista de patentes.
¿Qué pasaría si la patente fuera de 6 dígitos, sin letras? Sería mucho más fácil por nuestro "acostumbramiento" al uso de números decimales. Simplemente al número dado le sumaríamos K, y devolvemos eso.
Pensemos entonces por qué eso funciona. Lo que pasa ahí es que cada posición representa una potencia de 10, indicando cuántos números deja a la derecha. La unidad de mil deja 1000 números a la derecha, desde 0 a 999.
Y el número de la izquierda es cuántos grupos de mil, o cien, o 10^x números dejamos pasar.
Si pensamos en esto mirando la patente $ABA555$, pensemos: Cuántos grupos enteros de 1000 números, los que usan los 3 dígitos de la derecha desde 000 hasta 999, dejamos pasar? Bueno, hasta AAA dejamos 0, hasta AAB dejamos 1, y así siguiendo, hasta dejar pasar 25 estando en AAZ, y 26 en ABA. Veamos que lo que estamos haciendo para contar los grupos es simplemente mover el conjunto de 3 letras como si fuera un número en base 26. ¡Y pensar de esa forma está buenísimo!
Si transformamos las letras en dígitos donde $A = 0, B = 1, ..., Z = 25$, el número que se forma por ejemplo con CDE es el número 234 en base 26, o sea que hasta ahí dejamos pasar $2 \cdot 26^2 + 3 \cdot 26 + 4 = 1434$ grupos de 1000 patentes. Entonces la patente $CDE000$ es la número 1434000 (indexando en 0). Finalmente, por ejemplo,

la patente $CDE798$ es la patente de índice 1434798.
Ahora que sabemos encontrar el índice de una patente dada, y con esta idea de los grupos de 1000 que dejamos pasar, cómo podemos saber qué patente está en la posición p ?
Muy similarmente, pensamos en cuántos grupos de 1000 pasaron; para eso simplemente dividimos por 1000 y tomamos el resultado entero de eso. Ese número, es el que vamos a tener que pasar a base 26 y escribir las letras. Y luego, para lo que sobra (o sea el resto del número p en la división por 1000), es la cantidad de patentes que pasan adentro de un grupito de 1000. Entonces por ejemplo para $p = 1434225$, al dividir por 1000 es como sacar los últimos 3 dígitos, y nos queda el 1434, que sabemos que al pasar a base 26 tendremos las letras CDE. Y para lo que queda, que sería el 225, significa que después de los varios grupos de 1000, tenemos que pasar 225 grupos más, llegando justamente a los dígitos 225.
Ahora que terminamos este tipo de patentes, recomiendo tomarse un tiempo para pensar en el otro tipo de patentes.

Veamos entonces cómo resolver esto. La idea va a ser similar, pero ahora tenemos tres grupos. Entonces pensamos, cada vez que movemos el grupo de las dos letras de la izquierda, ¿cuántas patentes pasamos? Bueno, ya hicimos una cuenta similar antes, la cantidad será $10 \cdot 10 \cdot 10 \cdot 26 \cdot 26 = 676000$. Entonces, por ejemplo la patente $BE000AA$ será la número $676000 \cdot (1 \cdot 26 + 4) = 20280000$.
Y por cada vez que movemos el número del medio (por ejemplo de 012 a 013, ¿cuántos grupos de la derecha movemos? $26 \cdot 26 = 676$. Entonces, dada una patente podemos hacer esto: Convertimos las letras de la izquierda en base 26, y multiplicamos eso por 676000. A eso le sumamos el número del medio multiplicado por 676. Y a eso le sumamos el número que se forme con las dos últimas letras, en base 26.
Y para ir de una posición a una patente hacemos lo siguiente: Dividimos al número por 676000, y el resultado entero es el número en base 26 que se forma con las letras de la izquierda. Al resto que nos quedó, lo dividimos por 676, y es entonces la cantidad de veces que fuimos de AA a ZZ con las últimas dos letras, entonces ese resultado entero es el número que se forma en el medio. Y a lo que nos quedó de todo, es lo que tenemos que mover las últimas dos letras, entonces lo convertimos en base 26 y son las letras que nos quedan al final.

Una muy buena idea que para mí es más difícil que venga a la cabeza de una, pero que habiendo pensado todo lo que vimos hasta acá puede surgir, es, en vez de pensar a las patentes como grupos de 'dos letras', o 'tres letras', o 'tres dígitos', es simplemente **ir de a un caracter**.
Entonces, ¿qué tendríamos que modificar? Simplemente es ajustar el número que decíamos como 'cuántas veces muevo lo de la derecha para mover uno de la izquierda'.

Y por ejemplo esta idea usada en las primeras patentes, quedaría: Dada la patente $CDE798$, veamos la primera letra: ¿Cuántas patentes hay que avanzar para mover una vez este caracter? $26 \cdot 26 \cdot 10 \cdot 10 \cdot 10 = 676000$. Veamos que en esta cuenta hay algo parecido a lo que hacíamos antes; en base 26 a la primera letra la multiplicábamos por 26^2, que es lo mismo que estamos haciendo ahora, pero agregamos el 1000 que antes también agregábamos pero separado. Y así sucesivamente, podemos pensar para cada caracter.

Con esta idea, ¡con el mismo código podemos resolver el problema para cualquier tipo de patente! Por supuesto que si fuera una patente muy larga los números deberían ser muy grandes y capaz que la computadora no los puede almacenar, pero mientras los números no sean tan grandes, con el mismo código resolvemos el problema para cualquier tipo de patente.
El código que voy a dejar es este, así vemos un código que reusa funciones para situaciones que al principio parecía que había que resolver por separado.

Algo importante que podemos pensar, es cómo chequear que nuestra solución está bien, o al menos tener más seguridad sobre nuestra solución.
Con este problema, y con muchos otros, en los que tenemos una solución sencilla, rápida de programar, y que sabemos que está bien, podemos, una vez que tenemos nuestro código, probar varios casos que no le lleven mucho tiempo a la compu y probar de ambas maneras, la que queremos mandar como nuestra solución y la otra, menos eficiente.
En este caso, podemos usar la 'fuerza bruta' que pensamos al principio: Muchos FORs uno adentro de otro.
Entonces, podemos tener una función que resuelva el problema "lento pero seguro", como planteamos arriba de todo; y otra que lo resuelve rápido, pero que queremos chequear que no le pifiamos en ningún numerito, que siempre puede pasar.
De esta forma, generamos por ejemplo varias de las primeras patentes, y le pasamos muchos K distintos, y ponemos el programa a correr y que nos verifique si ambas funciones devuelven los mismo: Si es así, pensamos que está todo bien y dejamos esta solución; si no, vemos qué pasó y la corregimos, y volvemos a correr este comparador.

Obviamente, como sabemos que una de las dos funciones es lenta, vamos a probar casos sin maldad, es decir no vamos a pasar valores de K muy grandes, aprovechando que los vamos a elegir por nuestra cuenta. Pero el juez online podría pasar valores de K grandes, y por eso es que queremos la solución lo más rápida posible.
Dejo un código que hace esto, testea y nos dice si está todo bien. Algo importante

es que al mandar la solución, no tenemos que estar corriendo el chequeo siempre en los casos de prueba, porque si le pifiamos ya está, no podemos hacer nada. Entonces comentamos el “comparador”, y mandamos la solución.

```
#include<iostream>
#include<vector>
#include<set>
#include<math.h>
#include<algorithm>
#include<string>
#include<map>
#include<queue>

using namespace std;

bool esDigito(char c) {
    return c >= '0' && c <= '9';
}

int getPosition(string patente) {
    int dejoALaDerecha = 1;
    int n = patente.size();
    for(int i=0; i<n; i++){
        if(esDigito(patente[i])) {
            dejoALaDerecha *= 10;
        } else {
            dejoALaDerecha *= 26;
        }
    }
    int position = 0;
    for(int i=0; i<n; i++) {
        int cuantosGruposPasaron;
        int divido;
        if(esDigito(patente[i])) {
            cuantosGruposPasaron = patente[i] - '0';
            divido = 10;
        } else {
            cuantosGruposPasaron = patente[i] - 'A';
```

```
            divido = 26;
        }
        dejoALaDerecha /= divido;
        position += cuantosGruposPasaron * dejoALaDerecha;
    }
    return position;
}

string getPatente(int position, string patenteEjemplo) {
    int aLaDerecha = 1;
    int n = patenteEjemplo.size();
    for(int i=0; i<n; i++) {
        if (esDigito(patenteEjemplo[i])) {
        aLaDerecha *= 10;
        } else {
            aLaDerecha *= 26;
        }
    }
    string patenteToReturn = "";
    for(int i=0; i<n; i++){
        if (esDigito(patenteEjemplo[i])) {
            aLaDerecha/=10;
            int moves = position/aLaDerecha;
            patenteToReturn += char('0' + moves);
        } else {
            aLaDerecha/=26;
            int moves = position/aLaDerecha;
            patenteToReturn += char('A' + moves);
        }
        position %= aLaDerecha;
    }
    return patenteToReturn;
}

string siguiente(string patente, int K){
    int currentPosition = getPosition(patente);
    int wantedPosition = currentPosition + K;
    return getPatente(wantedPosition, patente);
}
```

```
string siguienteLentoPeroSeguroPatente1(string patente, int K) {
    bool alreadySeenMine = false;
    for(char c1=patente[0]; c1<='Z'; c1++){
        char c2;
        if (c1==patente[0]){
            c2 = patente[1];
        } else {
            c2 = 'A';
        }
        for(; c2<='Z'; c2++){
            char c3;
            if (c1==patente[0] && c2==patente[1]){
                c3 = patente[2];
            } else {
                c3 = 'A';
            }
            for(; c3<='Z'; c3++){
                char n1;
                if (c1==patente[0] && c2==patente[1] && c3==patente[2]){
                    n1 = patente[3];
                } else {
                    n1 = '0';
                }
                for(; n1<='9'; n1++){
                    char n2;
                    if (c1==patente[0] && c2==patente[1] &&
                        c3==patente[2] && n1 == patente[3]){
                        n2 = patente[4];
                    } else {
                        n2 = '0';
                    }
                    for(; n2<='9'; n2++){
                        char n3;
                        if (c1==patente[0] && c2==patente[1] &&
                            c3==patente[2] && n1 == patente[3] &&
                        n2 == patente[4]){
                            n3 = patente[5];
                        } else {
```

```
                        n3 = '0';
                    }
                    for(; n3<='9'; n3++){
                        if (alreadySeenMine) {
                            K--;
                            if(K == 0){
                                string ret = "";
                                ret += c1;
                                ret +=c2;
                                ret +=c3;
                                ret +=n1;
                                ret +=n2;
                                ret +=n3;
                                return ret;
                            }
                        }
                        if(c1==patente[0] && c2==patente[1] &&
                            c3==patente[2] && n1==patente[3] &&
                            n2==patente[4] && n3==patente[5]) {
                            alreadySeenMine = true;
                        }
                    }
                }
            }
        }
    }
    return "-1";
}

void testSolution() {
    for(char c1='A'; c1<='C'; c1++){
        for(char c2='A'; c2<='C'; c2++){
            cout<<c2<<endl; // Imprimo aca para ir viendo que avanza
                            // Puede tardar varios segundos
            for(char c3='A'; c3<='Z'; c3++){
                for(char n1='0'; n1<='3'; n1++){
                    for(char n2='0'; n2<='3'; n2++){
                        for(char n3='0'; n3<='3'; n3++){
```

```
                            string patente = "";
                            patente += c1;
                            patente +=c2;
                            patente +=c3;
                            patente +=n1;
                            patente +=n2;
                            patente +=n3;
                            for(int K = 1; K <= 100; K++) {
                                string lenta;
                                lenta = siguienteLentoPeroSeguroPatente1(patente, K
                                string answerPosta = siguiente(patente, K);
                                if (lenta != answerPosta) {
                                    cout<<"Para la patente "<<patente;
                                    cout<<" y K="<<K<<", deberia ser ";
                                    cout<<lenta<<" pero tu rta da ";
                                    cout<<answerPosta<<endl;
                                    cout<<"Posicion de la patente segun funcion: ";
                                    cout<<getPosition(patente)<<endl;
                                    return;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}


// ACA AGREGO EL EVALUADOR LOCAL
// ETO ESTA EN EL JUEZ, PERO LO DEJO ACA PARA QUE PUEDAN
// CORRER TODO EL CODIGO JUNTO EN SUS COMPUS

int main() {
    testSolution(); cout<<"Termine de testear"<<endl; // COMENTAR
    string patente;
    int K;
    cin>>patente>>K;
```

```
    cout<<siguiente(patente, K)<<endl;
}
```

4.1.3. Problema 3: Caminando al Colectivo [colectivo]

http://juez.oia.unsam.edu.ar/#/task/colectivo/statement

Este era el problema más difícil del certamen nacional 2019 en el nivel 1. Es un problema en el cual comparativamente no es tan difícil la implementación de la solución en la computadora, sino que la gran dificultad está en el análisis en papel y modelado del problema para entender bien la situación, y saber qué se debe calcular.

4.1.3.1. Subtarea 1

En esta subtarea hay un solo pueblo y una sola parada. Al haber una única parada, el pueblerino no tiene más opción que ir hasta ella. Así que podemos verificar dada la posición a del pueblerino y b de la parada, si $a < b$, en cuyo caso la respuesta es `E`, y sino la respuesta `O`. Esto puede hacerse con un único if en el código.

Esta implementación del if obtiene 2 puntos.

4.1.3.2. Subtarea 2

En esta subtarea hay muchos pueblerinos, pero todavía hay una única parada. La clave es que los pueblerinos no se afectan entre sí, así que como en la primera subtarea, cada uno tiene una única opción posible a donde ir, y basta ver si la única parada que existe queda al este o al oeste de cada uno de ellos. Por lo tanto, agregando un for a la solución de la primera subtarea para recorrer todos los pueblerinos y calcular el valor para cada uno, obtenemos una solución a esta otra subtarea.

Con esto se obtienen 9 puntos en total.

4.1.3.3. Caso general

Hay más subtareas que permiten enfoques especiales, pero explicaremos a continuación un enfoque general. Las subtareas permiten simplificar el análisis en general o tomar solo algunos de estos elementos y aún así poder obtener puntos.

En este problema existen colectivos viajando a velocidad uniforme. Siempre que tenemos una entidad que se mueve a velocidad constante en línea recta, podemos utilizar la expresión $\Delta e = v\Delta t$, donde Δe indica la distancia que se recorre en un tiempo Δt, cuando la velocidad constante es v.

La expresión anterior puede mejorarse para indicar a qué distancia estará un colectivo del origen en cada instante de tiempo t. Para esto, es necesario fijarnos primero un sistema de referencia bien definido, para que todos nuestros cálculos sean consistentes y nuestro programa arroje resultados correctos.

Tomaremos como inicio $t = 0$ el momento preciso en que los pueblerinos inician su caminata, que por enunciado sabemos que coincide casualmente con el momento en que un cierto colectivo sale de la terminal. Notemos que como los colectivos salen de la terminal periódicamente, quiere decir que en $t = -F$ había salido un colectivo, y otro en $t = -2F$ y así siguiendo, y que los próximos saldrán en $t = F$, en $t = 2F$ y así. En otras palabras, podemos numerar a todos los colectivos con un número entero n, y de esa manera el colectivo n sale de la terminal exactamente en el momento $t = nF$. El valor $n = 0$ corresponde al colectivo que sale de la terminal exactamente en el mismo instante en que los pueblerinos inician su caminata, los n negativos corresponden a los colectivos que salieron de la terminal antes de eso (los tiempos negativos), y que por lo tanto ya están en la ruta viajando al comenzar en $t = 0$. Y los n positivos corresponden a los colectivos que saldrán más adelante.

Es muy útil escribir una expresión que nos indique, para cada colectivo, en qué posición de la ruta se encuentra en el instante t. Para eso nos valdremos de dos datos:

1. En el instante nF, el colectivo n está en la ubicación 0 (la terminal)
2. Todos los colectivos viajan a velocidad v, así que cumplen $\Delta e = v\Delta t$.

Del segundo, podemos deducir que luego de t minutos, los colectivos viajarán una distancia adicional vt, y por lo tanto agregando esos t minutos al primer dato, sabemos que en el instante $nF + t$ el colectivo n se encuentra en $0 + vt = vt$.

Además, también por el segundo dato, en un tiempo nF los colectivos recorrieron una distancia vnF. Por lo tanto si a los tiempos anteriores le restamos nF, la posición de los colectivos estará vnF más atrás en ese mismo tiempo. De esta forma, en tiempo $(nF + t) - nF = t$ el colectivo n estará en $vt - vnF = v(t - nF)$. Esta es la expresión que buscamos, que nos dice la posición exacta de cada colectivo en cada instante.

En realidad, la expresión anterior asume que v, t y F están expresados con las mismas unidades, pero en el problema nos dan F en minutos, el t lo calcularemos en minutos, pero todas las velocidades v vienen en kilómetros por hora. El valor de la velocidad en kilómetros por minuto será $\frac{v}{60}$ (ya que en un minuto se recorre 60

veces menos distancia que en una hora), así que la expresión que nos dice en qué kilómetro de ruta está el colectivo n en el tiempo t es finalmente $\frac{v(t-nF)}{60}$.

Similarmente, si tenemos un pueblerino ubicado inicialmente a distancia p del origen, y que se desplaza a una velocidad de v_p kilómetros por hora (por lo tanto, $\frac{v_p}{60}$ kilómetros por minuto), podemos considerar solamente dos escenarios relevantes:

1. El pueblerino viaja al oeste. Entonces, en un tiempo de t minutos su ubicación será $p - \frac{v_p t}{60}$
2. El pueblerino viaja al este. Entonces, en un tiempo de t minutos su ubicación será $p + \frac{v_p t}{60}$

Supongamos que la siguiente parada al oeste está en A, y la siguiente parada al este está en B. Esto puede calcularse para cada pueblerino directamente recorriendo todas las paradas, y tomando como A la máxima entre aquellas que están en ubicaciones menores que p, y como B la mínima entre aquellas que son mayores que p. Si alguna de las dos no existe, porque todas las paradas quedan del mismo lado del pueblerino, tiene una sola opción hacia donde caminar, y la respuesta queda determinada automáticamente hacia ese lado, tal como hacíamos en las subtareas anteriores.

En el caso 1, el pueblerino camina hasta A y por lo tanto al oeste. En el tiempo t en que llegue allí, tenemos $A = p - \frac{v_p t}{60}$, entonces $t = \frac{60(p-A)}{v_p}$ es el tiempo exacto en que llega hasta A. De manera análoga será $t = \frac{60(B-p)}{v_p}$ el tiempo exacto en que llega hasta B si decide caminar al este.

Lo que tenemos que calcular ahora es cuál es el número del primer colectivo al cual se puede subir en cada caso, dado que llega a las paradas en esos tiempos. Si el número del primer colectivo es menor en el caso de A, la respuesta será `O`. Si es menor para B, será `E`. Y si en ambos casos se logra subir al mismo número de colectivo, la respuesta será `I` porque al subir al mismo colectivo, llegará a la capital en el mismo tiempo. Notar que esto puede pasar incluso si llega mucho antes a alguna de las dos paradas, pues si en ambos casos tiene que quedarse esperando hasta que pase el mismo colectivo, llegar antes no le gana nada.

Por los cálculos anteriores, sabemos que cuando el colectivo n llegue a la parada en A se tiene $A = \frac{v(t-nF)}{60}$, y de aquí que $t = \frac{60A}{v} + nF$ es el momento en que el colectivo n llega hasta A. Análogamente $t = \frac{60B}{v} + nF$ es el momento en que el colectivo n llega hasta B. El pueblerino puede alcanzar al colectivo n en A, si el tiempo que tarda en llegar allí, que ya calculamos antes, es menor o igual que el

tiempo en el cual el colectivo estará en A (de lo contario, el colectivo ya habrá pasado por A antes de que el pueblerino llegue).

Para esto tiene que ser $\frac{60(p-A)}{v_p} \leq \frac{60A}{v} + nF$. Para asegurarnos de operar en enteros y no tener ningun tipo de errores por la división, conviene multiplicar todo por $v_p \cdot v$ obteniendo la condición $60(p-A)v \leq 60Av_p + nFvv_p$, que utiliza los datos de la entrada y opera completamente en enteros. Similarmente podemos verificar si el pueblerino puede alcanzar al colectivo n en B verificando si se cumple la condición análoga $60(B-p)v \leq 60Bv_p + nFvv_p$.

Gracias a lo anterior, ya sabemos verificar dado un n, si ese colectivo n es alcanzable. Para saber el primer número de colectivo posible, podríamos hacer un despeje muy cuidadoso tomando parte entera, pero es más simple y seguro ir tanteando:

1. Iniciamos con $n = 0$
2. Mientras que el valor n actual no cumpla la condición, lo aumentamos en 1, para probar a ver si es posible alcanzar el siguiente colectivo.
3. Mientras que el valor $n-1$ también cumpla la condición, decrementamos n en 1 (esto para cubrir el caso en que de hecho el menor número de colectivo alcanzable es negativo: una alternativa para no hacer esto esto es inicializar con un número entero negativo de módulo suficientemente grande, en lugar de 0).

Implementando esta idea se obtienen 100 puntos.

4.2. Nivel 2

4.2.1. Problema 1: Escalera al cielo [escalera]

http://juez.oia.unsam.edu.ar/#/task/escalera/statement

Muchas veces sirve empezar pensando una solución sencilla, por más que no sea eficiente, de resolver el problema.
En general, lo más fácil para resolver un problema como este es simplemente simular toda la situación, en este caso subida y bajada de escalones.
Entonces podemos empezar con una variable **escalonActual** que empiece en 0, y hacemos por ejemplo un **WHILE(escalonActual <H)** que constantemente haga un FOR por los números de los elementos, sume o reste según corresponda, y mientras sume uno a la variable que cuenta la cantidad de pasos. Cuando pasamos

H devolvemos la cantidad de pasos y listo, sin hacer nada rebuscado.

Pensemos cuántas veces vamos a recorrer todos los números como mucho.

Según el enunciado, eventualmente llegamos a H. Entonces, podemos haber bajado al final de recorrer todos los números de v? Si bajamos, lo que pasa después es que volvemos a empezar, como al principio, pero ahora estaríamos más abajo. Entonces, si no llegamos antes a H, ahora peor. Y lo mismo cuando volvamos a completar la vuelta. Y esto mismo pasa si al terminar una vuelta volvimos al escalón inicial, nunca llegaríamos a H.

Entonces, asumiendo que eventualmente llegamos a la altura H, sabemos que al terminar una vuelta por lo menos subimos un escalón.

Por lo tanto, haciendo este proceso de simular todas las subidas y bajadas, a lo sumo cumpliremos H vueltas entera.

Si H es muy grande capaz no podamos hacer esta simulación. Pero si vemos la primera subtarea, N y H son a lo sumo 1000, por lo que recorrer H veces los N números, serían $H * N$ operaciones, y con esta solución obtendríamos los 25 puntos de la primera subtarea.

Ahora, es interesante que para las siguientes subtareas, $N = 1$ y $N = 2$, ya esta solución deja de andar, porque H puede ser muy grande,

Entonces, pensemos, si hay un solo número, qué podemos decir? Ese número tiene que ser positivo. Entonces, haciendo la cuenta matemática de $H/v[0]$ tenemos la cantidad de 'vueltas' que vamos a dar, que en este caso es un único número. Entonces por ejemplo si $H = 10$ y el número es 2, daremos 5 vueltas hasta llegar a H. Pero si el número es 3, haciendo 3 vueltas, llegamos a altura 9, y necesitamos subir una vez más esos 3 escalones. Si la división que mencionamos no da un resultado entero, y tomamos 'el resultado entero' de esa división, entonces ese número multiplicado por $v[0]$ será menor a H, por lo que nos faltará una vuelta más.

Si $N = 2$, es parecido, pero hay un par de cosas más que hay que considerar. Si con el primer salto llega a H, listo, la respuesta es 1. Si con los primeros dos saltos llega, será 2. Ahora, si no llega con esos dos saltos, volvemos a estar al principio, desde otro escalón. Recordemos que como sabemos que eventualmente llega, ya habíamos visto que al final de una vuelta debe haber subido.

Supongamos que por ejemplo, como antes, que $H = 10$, y al final de la primera subimos dos escalones. Podemos pensar que necesitaremos 5 vueltas exactas, por lo que 10 pasos serían los necesarios. Ahora, qué pasa si los números son por ejemplo $8, -6$? Ese es un caso en el que terminamos en 2 luego del primer ciclo, pero con una subida más ya estamos.

Propongo acá pensar, qué vamos a tener que mirar aparte de 'el escalón donde terminamos después del primer ciclo'.

Algo que nos interesará para saber cuándo llegaremos a H, es 'empezando por el primer número, a qué punto más alto puedo llegar'. Por qué? Porque supongamos que empezamos un ciclo en un escalón E. Luego, si a lo largo de un único ciclo podemos subir x escalones, entonces si $E + x >= H$, durante este ciclo en cuestión terminaremos el movimiento. Y si no, vamos a necesitar más de un ciclo. Y el próximo ciclo dónde estaremos? Bueno, si la suma total de cada ciclo es y, estaremos en $E + y$, y ahí arrancamos de vuelta.
O sea que lo vamos haciendo es ir al escalón y, $2y$, $3y$, y así, estando al inicio del ciclo en cada vez, hasta que llegamos a un múltiplo de y, digamos $n * y$, que hace que $n * y + x >= H$, donde n es el número de ciclos completos que hicimos.
Entonces, básicamente queremos el múltiplo de y tal que $n * y >= H - x$. Y los valores de H y x son constantes durante todo el procedimiento de subidas y bajadas.
Entonces, basta con calcular x e y, hacer $n = (H - x)/y$, aunque en realidad a eso hay que sumarle 1 si $(H - x)$ no es divisible por y, ya que queremos que $n * y >= H - x$. Si fuera divisible entonces el valor de n nos da la igualdad, pero si no, no nos alcanza con tomar la parte entera. Así sabremos que al menos n ciclos enteros vamos a necesitar.
Y sabemos que con $n + 1$ ciclos completos nos pasamos, porque ya llegamos a $n * y$ y en algún momento del último ciclo subiremos x escalones más.
Y cómo sabemos en qué posición del $n + 1$-ésimo ciclo llegaremos a H? Bueno, no hace falta que todo sea UNA cuentita! Ahora que lo reducimos a saber que estamos en el principio del ciclo en el escalón $n * y$, y que vamos a llegar en este próximo ciclo, podemos simular un ciclo. Es decir, al principio simular lleva mucho tiempo, pero nos queda mirar a lo sumo un ciclo más. Hacemos esto y terminamos.

```
long long int escalera(vector<long long int> v, long long int H) {
    long long int x=0, y=0;
    int n = v.size();
    long long int escalonActual = 0;
    for(int i=0; i<n; i++){
        escalonActual += v[i];
        x = max(escalonActual, x);
        y += v[i];
    }
    long long int ciclos = (H-x)/y;
    if (ciclos < 0) {
        ciclos = 0;
```

```
    }
    if (ciclos*y < H-x){
        ciclos++;
    }
    escalonActual = ciclos*y;
    long long int pasos = ciclos*n;
    if (escalonActual >= H){
        return pasos;
    }
    for(int i=0; i<n; i++){
        escalonActual += v[i];
        pasos++;
        if (escalonActual >= H) {
            break;
        }
    }
    return pasos;
}

// ACA DEJO EL MAIN DEL EVALUADOR LOCAL
// EL JUEZ YA LO TIENE PERO LO AGREGO PARA QUE PUEDAN TESTEAR EN SUS COMPUS

int main (){
    int n;
    long long int H;
    cin>>n>>H;
    vector<long long int> v(n);
    for(int i=0; i<n; i++){
        cin>>v[i];
    }
    cout<<escalera(v, H)<<endl;
}
```

4.2.2. Problema 2: Aplicando operaciones [aplicando]

http://juez.oia.unsam.edu.ar/#/task/aplicando/statement

Primero, veamos que N puede ser tan grande que lo vamos a guardar como string. Y recordemos que para formar un número entero a partir de un string de dígitos basta con empezar una variable en 0, e ir sumando los dígitos desde la

izquierda y multiplicando por 10.

Ahora, veamos las subtareas. Para $D = 1$, ¡el resto de dividir por 1 es siempre 0! Entonces basta con devolver siempre 0 para tener estos 3 puntos.

Con $D = 2$ y $D = 8$, basta saber que las 'reglas de divisibilidad' por estos números involucran la última o las 3 últimas cifras de un número. Entonces, sin importar $a[i]$ y $b[i]$, nos quedamos con las últimas tres cifras, formamos el número (que será chico, a lo sumo 999), y aplicamos la operación $num\%D$. Para cada pregunta hacemos poquitas operaciones porque manejamos a lo sumo 3 dígitos.
$D = 3$ ó $D = 9$ es un poco tramposa. Parece fácil porque lo que nos interesa es la suma de los dígitos, pero no lo es tanto. Tenemos que poder resolver el problema: Dado un arreglo de números, me tenés que poder responder cuánto vale la suma entre $a[i]$ y $b[i]$, para varios a, b distintos. Ese es otro lindo problema cuya idea nos va a servir en un futuro. Una manera de resolver este nuevo problema es guardarnos en un arreglo las 'sumas parciales', es decir primero guardamos la suma desde la posición 0 hasta la posición 0, después hasta la posición 1, y así siguiendo. Entonces, cuál es la suma entre la posición 4 y la posición 7? Podemos pensar que es la suma desde la posición 0 hasta la 7, restado la posición 0 hasta la 3, no? La cuenta quedaría $v[0] + v[1] + v[2] + v[3] + v[4] + v[5] + v[6] + v[7] - v[0] - v[1] - v[2] - v[3] = v[4] + v[5] + v[6] + v[7]$.

Entonces, calculando al principio de todo (antes de recibir las Q líneas) las sumas parciales (esto se llama “precomputar las sumas parciales”), podemos responder muy rápido la suma de cualquier rango de números consecutivos, haciendo la resta de dos valores que ya tenemos.
Para $N \leq 1{,}000{,}000, Q \leq 1000$ es muy parecido a $D = 8$, en el sentido que podemos para cada pregunta formar el número que nos están marcando, que tendrá a lo sumo 7 dígitos. Entonces para cada pregunta miramos a lo sumo 7 dígitos. De hecho, Q podría ser tan grande como 300,000, y estábamos bien. Construimos el número que nos piden en cada pregunta y hacemos $num\%D$, y listo.
Para N de mil dígitos, y $Q \leq 1000$, la única complejidad que tiene es manejar números muy grandes. Ahora no podemos formar el número del cual nos preguntan el resto, y hacer la cuenta. Para resolver esto, debemos saber o recordar esto: El resto de la división de $M * x$ dividido D, es el mismo que el resto de M dividido D, multiplicado por x, y a eso tomarle resto en la división por D. Es decir que básicamente si estamos buscando el resto en la división por D, en vez de M podemos tener el resto de M/D.

Veamos por qué: Si $M = D \cdot n + r$, con $0 \leq r < D$. Luego $M \cdot x = D \cdot n \cdot x + r \cdot x$, y si ahora $r \cdot x = D \cdot p + r_2$, con $0 \leq r_2 < D$, tenemos que $M \cdot x = D \cdot (n \cdot x + p) + r_2$,

por lo que vemos que el resto de $M \cdot x$ es el mismo que el de $r \cdot x$, que es r_2.
¿Por qué esto es útil?: Porque si vamos a tener un número de muchísimas cifras, nos interesa en vez de formar el número completo, ir quedándonos con el resto en la división por D. Queda verificar que si sumamos un número con otro podemos también quedarnos con el resto y listo.

Entonces, para esta subtarea, hacemos eso: Para cada una de las Q preguntas, vamos desde $a[i]$ hasta $b[i]$, hacemos lo de multiplicar por 10 y sumar cada dígito de izquierda a derecha, pero nos vamos quedando en cada paso con el resto en la división por D. Entonces para cada pregunta hacemos aproximadamente $b - a$ operaciones, que será menor o igual a 1000.

Para la subtarea de $a[i] = 1$, podemos usar la idea que vimos arriba de "precomputar": Al principio, empezamos por el primer dígito y vamos multiplicando por 10, sumando cada dígito, haciendo el resto en la división por D, y guardamos ese valor marcando que en la posición en la que estamos, tenemos el resto encontrado. Esto nos lleva $cifras(N)$ operaciones. Y después, cuando nos dicen el valor de $b[i]$, sin hacer ninguna operación devolvemos el valor que tenemos precomputado en esa posición.
Si bien parece que llegamos a esta de 'precomputar' de manera rápida y fácil, es un concepto clave para reducir el tiempo en muchas ocasiones.
Para resolver el problema por completo con las restricciones dadas, una idea es pensar lo siguiente: El número $A_3A_4A_5$ no es otra cosa que el número $A_1A_2A_3A_4A_5$, restado con A_1A_2000.
Entonces, por argumentos similares a los que venimos nombrando, si vamos a hacer sumas, restas, y multiplicaciones, y al final calcular el resto de la división por D, podemos quedarnos con los restos en los pasos previos y llegaremos al mismo resultado.
Por esta razón expliqué la subtarea previa, porque ahora podemos hacer lo mismo: Precomputamos los restos de los prefijos de N (es decir, empezando por $a = 1$). Entonces, el resto del subnúmero desde $a[i]$ hasta $b[i]$ es el resto desde la posición 1 hasta $b[i]$, menos el producto del resto del subnúmero desde 1 hasta $a[i] - 1$ con el resto de 10^{b-a+1}. Entonces otra cosa que tenemos que precomputar son los restos de todas las potencias de 10, para no tener que recalcularlos cada vez.

```
#include<iostream>
#include<vector>
#include<set>
```

```
#include<math.h>
#include<algorithm>
#include<string>
#include<map>
#include<queue>

using namespace std;

vector<int> dominando(string N, int D, vector<int> a, vector<int> b) {
    int len = N.size();
    vector<long long int> restosDesdePrimerDigito, restosPot10;
    restosDesdePrimerDigito.push_back(0);
    // Como empieza en 0, el resto hasta la posicion 1
    // sera restosDesde[1] y no tengo que ajustar los indices
    restosPot10.push_back(1); // Aca lo mismo, me queda en el indice 0
                            //  el resto de 10^0
    for(int i=0; i<len; i++) {
        restosDesdePrimerDigito.push_back((restosDesdePrimerDigito[i]*10+
                                            int(N[i]-'0'))%D);
        restosPot10.push_back((restosPot10[i]*10)%D);
    }
    vector<int> ans;
    int Q = a.size();
    for (int i=0; i<Q; i++) {
        int ai = a[i];
        int bi = b[i];
        ans.push_back(((restosDesdePrimerDigito[bi]-
            (restosDesdePrimerDigito[ai-1]*restosPot10[bi-ai+1])%D)+D)%D);
        // Sumo D porque como la resta puede dar negativa, para manejar siempre num
    }
    return ans;
}

// ACA DEJO EL CODIGO DEL EVALUADOR LOCAL, QUE YA VIENE EN EL JUEZ
// LO AGREGO POR SI QUIEREN PROBAR ESTE CODIGO ENTERO Y CORRERLO EN SUS COMPUS

int main() {
    string N;
    int D;
```

```
    cin>>N>>D;
    int Q;
    cin>>Q;
    vector<int> a, b;
    for(int i=0; i<Q; i++) {
        int a_i, b_i;
        cin>>a_i>>b_i;
        a.push_back(a_i);
        b.push_back(b_i);
    }
    vector<int> ans = dominando(N, D, a, b);
    for(int i=0; i<Q; i++){
        cout<<ans[i]<<endl;
    }
}
```

4.2.3. Problema 3: Truco de magia [mago]

http://juez.oia.unsam.edu.ar/#/task/mago/statement

El problema nos dice que hay N vasos numerados con bolitas adentro de cada uno. Nos dan una lista de pares (a_i, b_i) que representan las operaciones permitidas. Una operación (a_i, b_i) consiste en intercambiar la cantidad de bolitas del vaso a_i con la cantidad de bolitas de b_i. El problema nos da la cantidad de bolitas iniciales de casa vaso (a la que llamaremos ci_i), la lista de operaciones permitidas, la cantidad final de bolitas que quiere en cada vaso (a la que llamaremos cf_i) y nos pregunta si es posible llegar de la configuración inicial a la final utilizando las operaciones permitidas la cantidad de veces que queramos. Además, si es posible, nos pide que elijamos a lo sumo N de las operaciones que nos dan tal que sigue siendo posible usando solo estas operaciones.

Empecemos por averiguar cuándo es posible. Para esto vamos a modelar el problema con un grafo. En este grafo, cada nodo es un vaso con el numero correspondiente del vaso y unimos dos vasos con una arista si tenemos la operación que involucra a estos dos vasos. Es decir, unimos el vaso i con el vaso j si la operación (i, j) está en la lista de permitidas.

Una vez que construimos este grafo, vamos a mirar sus componentes conexas. La primera observación es que si miramos las cantidades de bolitas que tiene cada

vaso de una componente como multiconjunto, se mantiene a lo largo del proceso, por más operaciones que hagamos. Un multiconjunto es una lista de valores donde no importa el orden. Es decir, siempre habrá la misma cantidad de vasos con cierta cantidad de bolitas como había en la configuración inicial (siempre mirando sólo los vasos de una misma componente). Por ejemplo, si inicialmente, en una componente de vasos, hay un vaso con una bolita, dos vasos con dos bolitas y tres vasos con tres bolitas siempre habrá un vaso con una bolita, dos con dos bolitas y tres con tres bolitas. Esto es fácil de ver porque siempre que hacemos una operación o bien no afecta a los vasos de la componente y no cambia el multiconjunto o bien intercambia dos cantidades de vasos de la componente y solo cambia el orden de los valores. Esto nos da un criterio necesario para llegar de la configuración inicial a la final que es chequear que para cada componente del grafo los multiconjuntos de cantidades iniciales de los vasos y finales coincidan.

Además, esto es suficiente, ya que mediante operaciones podemos lograr cualquier asignación posible de las cantidades del multiconjunto en los vasos de la componente. Por ejemplo, si inicialmente hay un vaso con una bolita, dos vasos con dos bolitas y tres vasos con tres bolitas, mediante operaciones podemos llegar a cualquier configuración con un vaso con una bolita, dos vasos con dos bolitas y tres con tres bolitas. Hay varias formas de ver esto, veamos una que nos ayuda con la segunda parte del problema que es tomar un árbol generador de la componente e ir moviendo la cantidad necesaria de bolitas hacia las hojas (pueden leer sobre árboles generadores acá: `http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/arbol-generador`). Es decir, tomamos una hoja i del árbol y llevamos cf_i bolitas hacia la hoja i. Para esto tomamos un vaso j de la componente que tenga cf_i bolitas (la condición necesaria nos asegura que hay alguno), como ambos vasos están en la misma componente conexa, hay un camino que los une en el árbol generador. Luego aplicando las operaciones del camino en el orden yendo de j hacia i, logramos que el vaso i tenga cf_i bolitas. Ahora que terminamos con el vaso i, nos olvidamos de él, lo borramos del árbol generador. Como el vaso i fue borrado del árbol, nunca más cambiaremos su cantidad de bolitas por lo que siempre se mantendrá con cf_i. Y como era una hoja, nos asegura que al borrarlo el árbol que queda sigue siendo conexo. Además, al sacar el vaso i como tiene cf_i bolitas, las cantidades de bolitas actuales en los vasos que quedan y las finales siguen coincidiendo como multiconjuntos. Repetimos el proceso con el árbol sin el vaso i, hasta que no queden más vasos en el árbol. Cuando terminamos, todos los vasos de la componente tendrán su cf_i correspondiente.

¿Cómo nos ayuda esto para la segunda parte? La demostración nos dice que alcanza con usar las operaciones de un árbol generador de la componente para cada

componente. Luego, basta devolver las operaciones correspondientes a estos árboles generadores. Como cada árbol generador tiene una operación menos que la cantidad de vasos de la componente, nos asegura que la cantidad de operaciones que usamos es menos que la cantidad total de vasos, que es N.

Resumen del algoritmo que resuelve el problema:

- Construir el grafo de vasos y operaciones ($O(N + K)$).
- Encontrar sus componentes conexas. Esto se puede hacer fácilmente con DFS: http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/componentes-conexas ($O(N + K)$).
- Para cada componente chequear que las cantidades iniciales y las cantidades finales coincidan como multiconjunto. (Esto se puede hacer eficientemente con un map o armando las listas de cantidades iniciales y finales y ver que al ordenarlas coincidan) ($O(NlogN)$)
- Si no coinciden para alguna componente, responder NO.
- Construir un árbol generador para cada componente. Esto se puede hacer con un DFS por componente, tomando las aristas con las que llegamos por primera vez a cada nodo ($O(N + K)$).
- Responder SI y devolver la lista de aristas de cada árbol generador ($O(N)$).

Como vemos, el algoritmo tiene una complejidad total de $O(NlogN + K)$.

4.3. Nivel 3

4.3.1. Problema 1: Dominando operaciones [dominando]

http://juez.oia.unsam.edu.ar/#/task/dominando/statement

Si las cifras del número no pudieran cambiar, el problema sería el mismo que el problema **Aplicando operaciones** de este mismo certamen pero nivel 2. Se puede resolver, como mencionamos ahí, guardando los restos de los prefijos del número N en la división por D, y cuando nos pregunten por un subnúmero, en $O(1)$ tendríamos la respuesta.
Ahora, para manejar las queries de “cambio de cifras”, vamos a usar una estructura que se llama *segment tree*. Como su nombre lo indica, es un árbol, que guarda información sobre segmentos.

Entonces, en vez de tener la información de los prefijos únicamente, vamos a tener la información de muchos segmentos. ¿Cuántos? Alrededor de $n \cdot \lg(n)$, si n es la cantidad de cifras de N. Vamos a tener un segmento de longitud n, dos segmentos de longitud $\frac{n}{2}$, 4 segmentos de longitud $\frac{n}{4}$, y así sucesivamente, hasta tener n segmentos de longitud 1. Para cada longitud, lo que hacemos es dividir a las n cifras en conjuntos disjuntos: Las primeras k cifras son un segmento, las siguientes k otro segmento, y así sucesivamente.

Para entender por qué eso es eficiente y funciona bien para modificaciones, pueden leer este artículo: http://wiki.oia.unsam.edu.ar/algoritmos-oia/estructuras/segment-tree.

Lo que guardaremos en este caso es, para cada nodo del árbol, el resto en la división del subnúmero que representa ese nodo, por D. La idea es que si tenemos dos subnúmeros consecutivos por ejemplo, y queremos el resto del subnúmero que resulta de pegar estos dos números (lo que sería el segmento padre), es fácil de calcular: El subnúmero compuesto será $S1 \cdot 10^k + S2$, donde $S1$ y $S2$ son los subnúmeros separados, $S1$ el de la izquierda, y k es la cantidad de dígitos. Entonces, si tenemos precalculados los restos de las potencias de 10 en la división por K, será una simple cuenta que habrá que hacer para calcular el nodo del padre.

Luego, para cada cambio, solamente deberemos cambiar el nodo que contiene como subnúmero a esta única cifra, luego su padre (con la cuenta que dijimos), luego al padre del padre, y así hasta llegar a la raíz, efectuando un total de $log(n)$ pasos (altura del árbol).
Notar que las queries de consulta no necesariamente las podremos obtener de un nodo que contenga a ese subnúmero: Por ejemplo si tenemos el número 1234 y nos preguntan por el 123, lo que tendremos en el árbol será un nodo para el subnúmero 12 y otro para el 3, pero usamos la cuentita de arriba que pega los dos números y listo. La clave es no tener todos los segmentos (ya que hay n^2), pero sí poder cubrir a cualquier intervalo con pocos nodos.

4.3.2. Problema 2: Laboratorio lleno de perros [laboratorio]

http://juez.oia.unsam.edu.ar/#/task/laboratorio/statement

La solución de este problema consiste en modelarlo como un grafo, donde cada baldosa (celda de la matriz que describe el laboratorio) es un nodo. Una forma fácil de asignar los números de los nodos es que la baldosa ubicada en la fila i y la columna j se identifique con el número $iM + j$.

En este modelo cada baldosa está conectada a las (como máximo) 4 adyacentes, es decir, a las que puede alcanzar moviéndose una baldosa hacia arriba, abajo, izquierda o derecha. A su vez, solo vamos a tener en cuenta las baldosas libres, es decir que el grafo que modela el laboratorio no contendrá los nodos correspondientes a baldosas bloqueadas.

Es decir:

- Baldosas $\to$ nodos
- Una baldosa es adyacente a otra $\to$ arista

Además, se considerará que todas las *baldosas entrada* que estén libres están conectadas entre sí.

Ahora podemos observar que la pregunta de cuántas baldosas son accesibles es equivalente a preguntar el tamaño de la componente conexa que contiene a las entradas libres (si no hay ninguna entrada libre porque todas ya fueron ocupadas por perros, entonces esta componente no existe y se considera de tamaño 0).

Por lo tanto, ya podemos reformular el enunciado como:

“Se tiene un grafo grilla, con una serie de nodos especiales (las entradas `E`) que están todos conectados entre sí. De este grafo se van eliminando nodos uno a uno. Después de cada eliminación, se debe indicar el tamaño de la componente conexa que contiene los nodos especiales.”

Es muy importante para la solución observar que este es un problema offline. Esto quiere decir que podemos leer la entrada y ver desde el comienzo todas las consultas ‘queries” que se nos hacen (nodos que se van borrando). La versión online del problema sería aquella en la cual nos van indicando los nodos que se borran uno a uno, y estamos obligado a responder cada consulta en el momento, sin conocer todavía las siguientes. La versión online es muchísimo más difícil, por lo cual en este problema es importantísimo aprovechar que conocemos todas las consultas, incluidas las últimas.

El hecho de conocer ya todas las baldosas donde se ubicarán los perros desde el comienzo permite pensar el proceso de manera inversa en el tiempo:

“Se tiene un grafo, y se van **agregando** nodos, de los cuales algunos pueden ser especiales, y esos están conectados todos entre sí. Cada vez que voy a agregar un nodo, quiero antes saber el tamaño de la componente conexa que contiene a los nodos especiales.”

El problema así planteado es mucho más simple porque admite la utilización de la estructura de datos eficiente Union-Find (http://wiki.oia.unsam.edu.ar/algoritmos-oia/estructuras/union-find). En este caso al agregar un nodo, agregamos también las aristas que lo unen a los nodos existentes adyacentes (a las baldosas adyacentes, y a algún otro nodo especial preexistente si el nodo agregado es especial), mediante la operación de unir los dos nodos extremos de la arista.

Algo conveniente es que podemos usar la misma estructura al comienzo, para unir entre sí los nodos adyacentes en el grafo inicial (que en el problema original, corresponde al resultado cuando ya están todos los perros ubicados). Esto evita calcular con un algoritmo diferente las componentes conexas al comienzo.

Repasando: inicialmente tendremos todos los perros ya ubicados en sus lugares (lo que corresponde al P-ésimo escenario, la última posición del vector que se debe retornar). En el grafo entonces no tendremos los nodos que corresponden a los perros. Desde allí iremos quitando los perros en el orden inverso al que entraron, y cada vez que hacemos eso, se agregará un nuevo nodo libre al grafo (**posiblemente una entrada**, la cual debe ser manejada con especial cuidado). Utilizando el algoritmo de Union-Find, podremos mantener la conectividad eficientemente.

Para poder obtener el tamaño de la componente rápidamente, debemos ampliar el algoritmo básico de Union-Find (que únicamente indica si dos nodos están conectados entre sí o no) con un arreglo de enteros que almacene por cada componente, su tamaño. Lo único que hay que hacer para mantener este arreglo es asegurarnos que, cuando se hace una unión de dos componentes, la componente resultante va a tener como tamaño la suma de los tamaños, y hacer esta actualización en el arreglo.

Algunas soluciones parciales existentes:

- Para la subtarea $P = 1$, una solución consiste simplemente en ubicar al perro en la correspondiente baldosa, bloqueándola. Luego se puede recorrer el grafo con BFS o DFS, para encontrar las componentes conexas y ver el tamaño de la que contiene entradas (si quedaron entradas sin tapar).
- Para la subtarea con $N \cdot M \cdot P$ pequeño se puede utilizar la misma idea, pero debe volverse a ejecutar un nuevo BFS o DFS para recorrer todo el grafo cada vez que se agrega un perro nuevo (lo que corresponde a sacar un nodo, marcando ahora la baldosa como bloqueada).
- En la subtarea $N = 1$ se puede realizar una observación fundamental: el laboratorio entero se reduce a una única fila, o sea que el problema queda

simplificado al caso de una única dimensión. En la fila habrá un conjunto contiguo de baldosas que son las entradas, es decir, existe un **intervalo** de baldosas entrada. Al ser una sola dimensión, cada obstáculo separa automáticamente todas las baldosas a izquierda de las baldosas a derecha. Por lo tanto, el laboratorio siempre está particionado en intervalos maximales de baldosas libres, separados entre sí por obstáculos (o perros que ingresaron). Podemos clasificar las baldosas inicialmente libres en 3 tipos: las de entrada, las que están antes de las de entrada, o las que están después de las de entrada (si recorremos de izquierda a derecha).
Así, cuando bloqueamos una baldosa que antes era accesible sucede que:

1. Si la baldosa está antes del intervalo de entrada, o es el extremo menor del mismo, deja inaccesible a todas las baldosas anteriores.
2. Si la baldosa está después del intervalo de entrada, o es el extremo mayor del mismo, deja inaccesible a todas las baldosas posteriores.
3. Si la baldosa está contenida en el intervalo de entradas y no es ninguno de sus extremos, no afecta a ninguna otra baldosa.

Es importante notar que en los supuestos 1 y 2, cuando estamos marcando todas las anteriores/posteriores como inaccesibles y encontramos una que ya está marcada, debemos cortar ahí no seguir recorriendo la fila, para garantizar una complejidad lineal $O(M)$ ya que cada baldosa se marcaría a lo sumo una vez.

4.3.3. Problema 3: Ayudando al Electricista [electricista]

http://juez.oia.unsam.edu.ar/#/task/electricista/statement

La solución presenta 3 dificultades importantes:

- Modelar correctamente el problema (visualizarlo)
- Identificar qué es lo que está pidiendo y con qué cuenta puede lograrse.
- Encontrar una forma eficiente de realizar dichas cuentas.

Respecto del primer ítem, puede verse de la siguiente manera (aunque no es necesario para resolver el problema):

“Dado un **Trie**, cuántos conjuntos de X palabras diferentes existen que formen dicho **Trie**”

Lo importante a la hora de modelarlo es tomar la noción de que los tubos y rosetas forman un grafo, un árbol con raíz específicamente:

- Las roseta, junto a la conexión central, son los nodos.
- La conexión central 0 es la raíz del árbol (que en el **Trie** representa la palabra vacía).
- Los tubos son las aristas del árbol.

Además, los cables son caminos que unen la raíz con alguno de los nodos (la roseta donde termina el cable), y los nodos solo existen si aparecen en al menos uno de los caminos. De lo anterior, se desprende que para cada hoja del árbol debe haber un camino que llegue hasta ella. Y además, con eso será suficiente para que existan todos los nodos.

Las observaciones anteriores se evalúan en la subtarea por 5 puntos donde todas las rosetas están conectadas a la roseta 0, es decir, únicamente existen las hojas y la raíz. También es necesario modelar mediante un árbol jerarquizado para resolver la subtarea de 17 puntos donde $N \leq 12$.

Para las siguientes subtareas será también necesario notar que se puede definir un cable únicamente por la roseta en la que termina. Es decir, una vez fijado el extremo del camino que es distinto de la raíz, al ser un árbol existe un único camino simple desde la raíz hasta ese nodo extremo.

La siguiente dificultad es identificar correctamente la pregunta y cómo calcular su respuesta.

Se debe notar que, como se dijo antes, debe haber un camino que una cada hoja con la raíz. Por lo tanto:

- Llamamos H a la cantidad de hojas del árbol.
- Notamos que existen nodos interiores, que no son ni la raíz ni una hoja. Llamamos I a la cantidad de nodos interiores. Se tiene que $I = N - H$
- Se observa que como existe necesariamente un cable hasta cada hoja, queda una cantidad $C = X - H$ de cables adicionales, que tendrán que ir hasta nodos interiores.

Notar que si $X > N$ o si $X < H$, es imposible armar la red.

Como una vez abarcadas todas las hojas ya está garantizada la existencia de todos los nodos (pues ya hay al menos un camino pasando por cada nodo), cualquier forma de distribuir los C cables adicionales entre los I nodos interiores será válida, mientras que no se haga terminar dos cables en un mismo nodo.

Por lo tanto, se debe calcular cuántas formas existen de tomar C elementos de un conjunto de I elementos. Esta observación se evalúa en la subtarea por 23 puntos donde $N \leq 1{,}000$.

Finalmente, el tercer problema es cómo implementar de forma eficiente este cálculo.

La operación es $\binom{I}{C} = \frac{I!}{C!(I-C)!} = \frac{I\cdot(I-1)\cdot(I-2)\cdots(I-C+2)\cdot(I-C+1)}{C!}$

Además, puede observarse que:

- Si se desean factorizar todos los números desde 1 hasta I, esto puede realizarse en tiempo $O(I \ln I)$ utilizando cualquier idea del estilo de la Criba de Eratóstenes `http://wiki.oia.unsam.edu.ar/algoritmos-oia/enteros/criba-de-eratostenes`.
- Multiplicar dos números implica sumar los exponentes de sus factores primos (al considerar las factorizaciones de los números). A su vez, dividirlos implica restar los exponentes del divisor a los del dividendo.

Teniendo esto en cuenta, lo que se puede hacer es:

- Definir un vector `cantf` que tendrá la factorización del resultado de la operación. `cantf[i]` contiene el exponente de i en la factorización de la respuesta.
- Recorrer todos los números entre I e $I - C + 1$, y para sus factores primos, sumar sus exponentes en `cantf`.
- Recorrer todos los números entre 1 y C, y para sus factores primos, restar sus exponentes en `cantf`.
- Finalmente, `cantf` contendrá la factorización del resultado. Es posible entonces calcular en una última iteración la respuesta operando modularmente, ya que será $(1^{cantf[1]} \cdot 2^{cantf[2]} \cdot 3^{cantf[3]} \ldots I^{cantf[I]})\ \%MOD$

La complejidad de esta solución es $O(I \ln I)$

Notar que al depender del MOD del caso de prueba, no es posible utilizar en forma directa soluciones basadas en el uso del inverso modular, excepto para la subtarea específica de $MOD = 2$, que aporta 25 puntos.

Es posible utilizar técnicas de inverso modular si se factoriza MOD, y se computan independientemente (de manera similar a lo que describimos para hacer con **todos** los primos) los exponentes para los primos que dividen al módulo. Sin embargo esta versión es probablemente más compleja y mucho más teórica que la aquí explicada. La complejidad final sería similar y permitiría resolver el problema, no obstante.

También es posible factorizar cada número en forma independiente, en lugar de realizar un precalculo con criba de Eratóstenes. Esto es más ineficiente, pero si está bien implementado, cortando la búsqueda de factores no bien se descubren todos los factores del numero, y probando hasta la raíz de los números como máximo, alcanza para resolver este problema en tiempo.

Algunas soluciones parciales que existen son:

- Calcular los combinatorios utilizando el triángulo de pascal, lo que da una complejidad $O(I^2)$. https://es.wikipedia.org/wiki/Triangulo_de_Pascal
- En la subtarea con $MOD = 2$, solamente hay que saber si la respuesta es par o no. Para esto no hace falta factorizar nada, sino simplemente contar cuántos factores 2 tiene cada número, lo que es mucho más simple que una factorización completa. De esta forma, contamos la cantidad de factores 2 en el numerador y en el denominador, y si la cantidad del numerador es mayor estricta, la respuesta es 0, y sino es 1.